DataFlow JavaBeans for Collaborative Social Network Analysis

Dissertation

submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in Social Science

by

Jeffrey Alan Stern

Dissertation Committee:
Professor Linton C. Freeman, Chair
Professor John P. Boyd
Sue C. Freeman, PhD
Professor A. Kimball Romney

2003
The dissertation of Jeffrey Alan Stern
is approved and is acceptable in quality
and form for publication on microfilm:

John P. Boyd

Sue C. Freeman

A. Kimball Romney

Linton C. Freeman, Committee Chair

University of California, Irvine
2003
Dedication

To Good Mentors
# Table of Contents

List of Figures ......................................................... xiii

List of Tables .......................................................... xxii

Acknowledgements ....................................................... xxiv

Curriculum Vitae ........................................................ xxvii

Abstract of the Dissertation ........................................... xxviii

1 Introduction ............................................................ 1

1.1 Background to the Problem ......................................... 1

1.2 The Problem .......................................................... 3

1.3 Modular Visualization Environments as Potential Solution .......... 13

1.3.1 Introduction to Modular Visualization Environments ............. 13

1.3.2 Architecture and Advantages of MVEs .......................... 16

1.3.3 Dataflow Architectures: Specific Advantages to Social Network Researchers ................................................. 19

1.3.4 Disadvantages of MVEs ........................................... 21

1.4 A Cross-platform Rewrite ........................................... 24

1.5 Java beans to the rescue ............................................ 27

1.6 GeoVista Studio .................................................... 29

1.7 Problems with JavaBeans .......................................... 32
1.8 JavaBeans extended: Ancestor classes ............................................ 35

1.9 Data types .................................................................................. 36

1.9.1 Introduction ............................................................................. 36

1.9.2 Tensor and GraphArray .............................................................. 37

1.9.3 Other Data Types ...................................................................... 38

1.10 Conclusion ................................................................................ 39

2 Installation ....................................................................................... 40

2.1 Install the Java 2 SDK .................................................................. 41

2.2 Install the socnet beans ................................................................. 42

2.2.1 Download the necessary files ................................................... 42

2.2.2 Tell your JVM where the jars are ............................................. 43

2.3 Install Java Web Start .................................................................. 44

2.3.1 Windows .................................................................................. 45

2.3.2 Linux ...................................................................................... 45

2.4 Install and Configure Geo Vista Studio ........................................ 49

2.4.1 Installing .................................................................................. 49

2.4.2 Configuring .............................................................................. 50

2.4.3 Import the Socnet Palette into Studio ..................................... 52

2.5 Installing Other Beans (e.g. ChemSymphony) ............................... 53

2.5.1 Install the ChemSymphony jar file ......................................... 54

2.5.2 Make a new Palette ................................................................. 54

2.5.3 Make a new Bean Folder ......................................................... 55

v
### 2.5.4 Import the beans

### 2.5.5 Select all beans

### 2.5.6 Manually configure the ChemSymphony beans

### 2.6 Installing External Software (e.g. Mage)

### 3 Pretutorial: Background to Using the Socnet Beans

#### 3.1 Introduction

#### 3.2 Understanding Object Oriented Programming Terminology

#### 3.3 Data Types and Method Declarations

- **3.3.1 Data Types**
- **3.3.2 Method Declarations**

#### 3.4 Understanding Beans

- **3.4.1 Setters and Getters for Properties**
- **3.4.2 The Socnet Bean Data Processing and Communication Cycle**
- **3.4.3 Connecting Socnet Beans**

### 4 Tutorial Part 1: Correspondence Analysis of Two-Mode data with the Socnet Beans

#### 4.1 Introduction

#### 4.2 Read In the Data

- **4.2.1 Create the input file**
- **4.2.2 Read in the input file**
- **4.2.3 Test that you’ve read in the DL file**

#### 4.3 Correspondence Analysis: Two-Mode
4.3.1 Normalize the data ........................................ 114
4.3.2 Perform the Singular Value Decomposition .......... 116
4.3.3 Scale the row and column scores .................... 119
4.3.4 Weight the Scores ...................................... 121
4.3.5 Stack the Row and Column Scores .................. 124
4.4 Convert to a General Visual Representation (GraphArray) ... 128
4.4.1 Convert Tensor to GraphArray ....................... 129
4.5 Visualize the data with Mage ............................ 133
4.5.1 Convert GraphArray to Mage ......................... 133
4.5.2 Save Mage output to a file ........................... 134
4.5.3 View data with Mage .................................. 140
4.6 Visualize the data with ChemSymphony ................. 144
4.6.1 Convert GraphArray to PDB String .................. 144
4.6.2 Direct PDB output to JTextArea window and to a file ... 146
4.6.3 Add ChemSymphony beans to design ................ 149
4.6.4 View data with ChemSymphony ...................... 152
4.7 Conclusion ................................................. 158

5 Tutorial Part 2: Correspondence Analysis of One-Mode data with the Socnet Beans .............................. 160
5.1 Introduction .................................................. 160
5.2 Setup ....................................................... 161
5.2.1 Create a new directory ............................... 161
5.2.2 Rename the design and save it to the new directory . . . . 161
5.2.3 Disconnect Correspondence Analysis from the design . . . 162
5.2.4 Rearrange the design . . . . . . . . . . . . . . . . . . . . . . 163

5.3 Read in the Data . . . . . . . . . . . . . . . . . . . . . . . . . . . 164
5.3.1 Create the input file . . . . . . . . . . . . . . . . . . . . . . . 164
5.3.2 Read in the input file . . . . . . . . . . . . . . . . . . . . . . 167
5.3.3 Verify data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 167

5.4 Reverse the data . . . . . . . . . . . . . . . . . . . . . . . . . . . 170
5.4.1 Reverse the data with a Reverse Tensor bean . . . . . . . . 171
5.4.2 Verify the Reversal . . . . . . . . . . . . . . . . . . . . . . . . 172

5.5 Symmetrize the data . . . . . . . . . . . . . . . . . . . . . . . . . 173
5.5.1 Symmetrize the data with a Symmetrize Tensor bean . . . . 174
5.5.2 Verify the Symmetry . . . . . . . . . . . . . . . . . . . . . . . 176

5.6 Collapse the data . . . . . . . . . . . . . . . . . . . . . . . . . . . 177
5.6.1 Collapse the data with a Collapse Tensor bean . . . . . . . . 178
5.6.2 Verify the matrices have been collapsed . . . . . . . . . . . 179

5.7 Reconnect Correspondence Analysis beans . . . . . . . . . . . . 181
5.7.1 Move ObjectToString bean . . . . . . . . . . . . . . . . . . . 181
5.7.2 Reconnect CA beans . . . . . . . . . . . . . . . . . . . . . . . . 181
5.7.3 Verify the Correspondence Analysis of the Collapsed Tensor 182

5.8 Expand the data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182
5.8.1 Expand the data with the Expand Tensor bean . . . . . . . . 184
5.8.2 Verify the Tensor’s matrix has been expanded . . . . . . . . 187
7.1 Introduction .......................................................... 251
7.2 Programming a Java Bean .......................................... 252
7.3 Background to Programming a FunctionBean .............. 253
  7.3.1 Class hierarchy of the Socnet beans ....................... 253
  7.3.2 Logging events with StackLogger ......................... 255
  7.3.3 The Data Processing Cycle of a FunctionBean .......... 259
  7.3.4 Multi-threading and synchronization ...................... 260
7.4 Making your FunctionBean ........................................ 263
  7.4.1 Set up core file, declare package, imports, and Class ... 263
  7.4.2 Plan your input and output properties .................... 265
  7.4.3 Declare your private variables ............................. 266
  7.4.4 Program your setters and getters .......................... 268
  7.4.5 Create a zero-parameter constructor ..................... 273
  7.4.6 Override readyToCalculate() ............................... 275
  7.4.7 Override calculate() ......................................... 279
  7.4.8 Write your core code ....................................... 282
  7.4.9 Override clearInputs() ..................................... 283
  7.4.10 Decide on Autocalculation ................................ 284
  7.4.11 Decide on AutoClearing of Inputs ......................... 286
  7.4.12 Decide on Logging Level ................................... 287
  7.4.13 Add an SUID: a Stream Unique Identifier .............. 288
7.5 Extending your bean’s functionality with a BeanInfo class ... 290
  7.5.1 Create the bean’s icons ..................................... 291
LIST OF FIGURES

1.1 OpenDX .................................................. 15

2.1 Java Web Start: Startup ................................... 48
2.2 Java Web Start: Choosing to view Downloaded Applications . . . 48
2.3 Java Web Start: GeoVista Studio as a Downloaded App ............... 49
2.4 GeoVista Studio run for the first time .................................. 50
2.5 Telling Studio where your Java Compiler is ............................. 51
2.6 Socnet Palette and Beans in GeoVista Studio ............................ 53
2.7 Selecting the ChemSymphony Jar File ................................... 55
2.8 Adding all the ChemSymphony beans to GeoVista Studio ......... 56
2.9 The ChemSymphony beans added to the palette ....................... 56
2.10 Using the mouse to find the RenderBasic ChemSymphony bean . 57
2.11 The Configure Bean button ........................................ 58
2.12 RenderBasic Properties Dialog: setting the three inputs ......... 59
2.13 The RenderBasic Bean with 2 input nodules (on left-hand side) . 60
2.14 Using the mouse to find the RenderControl ChemSymphony bean . 60
2.15 Setting the RenderControl Output notification in the Properties Dialog ........................................ 61
2.16 Setting the RenderControl Callback in the Properties Dialog .... 62
2.17 Mage .................................................. 63

3.1 A simple class hierarchy for PropPlane .................................. 69
3.2 PropPlane, String, and Vector, as descendants of Object... 72
3.3 DL descends from (is a) 'Tensor' ............................. 73
3.4 Studio uses Java-style method declarations to communicate what
data will be sent or received ................................. 75
3.5 Studio connects beans with an invisible adapter .............. 84
4.1 DL-formatted SES/Mental data ............................. 87
4.2 ReadDL bean placed onto DesignBox. Marker automatically placed
    in GUIBox .................................................. 89
4.3 GUIBox: The MtSimpleFileChooser bean’s GUI appearance.... 90
4.4 DesignBox: Finding the doCalculate() input node on the ReadDL
    bean ....................................................... 91
4.5 DesignBox: Finding the setInputDLFileName(String) input node
    on the ReadDL bean ....................................... 91
4.6 DesignBox: Finding the propertyChange output node on the ReadDL
    bean ....................................................... 92
4.7 DesignBox: Finding the actionPerformed() output node on the Mt-
    SimpleFileChooser bean ................................... 93
4.8 DesignBox: Connecting MtSimpleFileChooser to ReadDL ...... 94
4.9 The Adapter Wizard for connecting outputs to inputs ....... 95
4.10 DesignBox: MtSimpleFileChooser and ReadDL connected .... 96
4.11 DesignBox: Connecting JButton to ReadDL .................. 96
4.12 The Adapter Wizard with no arguments required .......... 97
4.13 DesignBox: JButton connected to ReadDL ................... 98

xiv
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.14</td>
<td>DesignBox: Accessing the properties for JButton</td>
<td>98</td>
</tr>
<tr>
<td>4.15</td>
<td>JButton Properties dialog: Entering a label in the Text property</td>
<td>99</td>
</tr>
<tr>
<td>4.16</td>
<td>GUIBox: JButton with new text label</td>
<td>99</td>
</tr>
<tr>
<td>4.17</td>
<td>DesignBox: ObjectToString and JTextArea added</td>
<td>101</td>
</tr>
<tr>
<td>4.18</td>
<td>JTextArea Properties Dialog Box</td>
<td>103</td>
</tr>
<tr>
<td>4.19</td>
<td>JTextArea Properties dialog: Input choices</td>
<td>104</td>
</tr>
<tr>
<td>4.20</td>
<td>JTextArea Properties dialog: Choosing to see the input methods for the ancestor class JTextComponent</td>
<td>105</td>
</tr>
<tr>
<td>4.21</td>
<td>JTextArea Properties dialog: Adding the setText input method from JTextComponent</td>
<td>106</td>
</tr>
<tr>
<td>4.22</td>
<td>DesignBox: JTextArea with second input connector added</td>
<td>106</td>
</tr>
<tr>
<td>4.23</td>
<td>DesignBox: Verifying the setText(String) method input connector</td>
<td>107</td>
</tr>
<tr>
<td>4.24</td>
<td>Selecting the JTextArea bean in the bean folder</td>
<td>107</td>
</tr>
<tr>
<td>4.25</td>
<td>Configuring all future instances of the JTextArea bean</td>
<td>107</td>
</tr>
<tr>
<td>4.26</td>
<td>DesignBox: First design finished</td>
<td>108</td>
</tr>
<tr>
<td>4.27</td>
<td>GUIBox/MtSimpleFileChooser: Input file chosen</td>
<td>109</td>
</tr>
<tr>
<td>4.28</td>
<td>GUIBox: First design run</td>
<td>110</td>
</tr>
<tr>
<td>4.29</td>
<td>DesignBox: Selecting a connection</td>
<td>111</td>
</tr>
<tr>
<td>4.30</td>
<td>DesignBox: Deleting a connection</td>
<td>111</td>
</tr>
<tr>
<td>4.31</td>
<td>Telling Studio you want to save your design</td>
<td>112</td>
</tr>
<tr>
<td>4.32</td>
<td>Studio's Save Design dialog box</td>
<td>112</td>
</tr>
<tr>
<td>4.33</td>
<td>DesignBox: Connecting the SqRtNorm bean to ObjectToString</td>
<td>115</td>
</tr>
<tr>
<td>4.34</td>
<td>GUIBox: Output of the SqRtNorm bean</td>
<td>116</td>
</tr>
</tbody>
</table>
4.35 Adapter Wizard: **SVD**’s available outputs (getters) ........................................ 118

4.36 GUIBox/JTextArea: Square-root normalized row scores, as a **Tensor** printing out as a **String** ................................................................. 119

4.37 DesignBox: **OptScale** added ........................................................................... 120

4.38 Adapter Wizard: The **OptScale** getters (outputs) ........................................ 122

4.39 GUIBox: **OptScale** output: Optimally scaled Row Scores .......................... 123

4.40 GUIBox: Weighted Row Scores ........................................................................ 125

4.41 GUIBox: Weighted Column Scores ................................................................... 126

4.42 DesignBox: **StackTensors (STKT)** bean added ........................................... 127

4.43 GUIBox: The Stacked Tensor of Row and Column Scores ............................ 128

4.44 DesignBox: **TensorToGraphArray (T2GA)** bean added ............................. 130

4.45 GUIBox/JTextArea: **GraphArray** as a **String** ............................................. 131

4.46 DesignBox: Right-clicking on **TensorToGraphArray** to open the
Properties dialog box ......................................................................................... 131

4.47 **TensorToGraphArray** Properties: Changing the **rowVertexLabel**
property ............................................................................................................. 132

4.48 GUIBox/JTextArea: **GraphArray** as a **String**, row labels corrected ........ 133

4.49 DesignBox: **GraphArrayToMage** added ....................................................... 135

4.50 GUIBox/JTextArea: Output of **GraphArrayToMage**: a kinemage **string** .......................... .............................................................................. 136

4.51 GUIBox: Configuring **MtSimpleFileChooser** style: Save instead of Open ............................. ................................. 137
4.52 MtSimpleFileChooser’s popup dialog box: Choosing an output file name ........................................... 138

4.53 DesignBox: 2-mode dataflow network design for creating a Mage image ............................................ 139

4.54 Mage: Open File dialog ........................................... 141

4.55 Mage: Viewing the 2-mode SES/Mental data ........................................... 142

4.56 Mage: Hiding the mental health (“set 1”) balls ........................................... 143

4.57 GUIBox/MtSimpleFileChooser: Changing the output filename for our PDB file ........................................... 147

4.58 DesignBox: GraphArrayToPDB (GA2PDB) added ........................................... 147

4.59 GUIBox/JTextArea: PDB output of GraphArrayToPDB ........................................... 148

4.60 GUIBox: Re-arranging the RenderBasic and RenderControl bean components ........................................... 151

4.61 DesignBox: Finished 2-mode dataflow pipeline design ........................................... 153

4.62 ChemSymphony RenderBasic: Srole SES-Mental data viewed for first time ........................................... 154

4.63 ChemSymphony RenderControl: Changing the Display Style to Balls ........................................... 154

4.64 ChemSymphony RenderControl: The SRole SES-Mental data rendered with the Balls Display Style ........................................... 155

4.65 ChemSymphony RenderControl: Reducing the size of the rendered atoms ........................................... 156

4.66 ChemSymphony RenderBasic: An improved view of the SRole SES-Mental data ........................................... 157
4.67 GUIBox: Final 2 mode data viewer running SRole SES-Mental data as PDB with ChemSymphony beans

5.1 Saving your design under a new name and directory

5.2 DesignBox: Deleting StackTensors

5.3 DesignBox: Beans rearranged to make room for new beans

5.4 Beginning of DL-formatted Newcomb fraternity data

5.5 DesignBox/MtSimpleFileChooser: Input file chosen for Newcomb fraternity data

5.6 DesignBox: ObjectToString added

5.7 GUIBox/JTextArea: The NewFrat data in Tensor format

5.8 DesignBox: ReverseTensor added

5.9 GUIBox/JTextArea: the Reversed Newfrat Tensor

5.10 Properties for SymmetrizeTensor: checking method is by SUM

5.11 DesignBox: SymmetrizeTensor added

5.12 GUIBox/JTextArea: the symmetrized newfrat Tensor

5.13 DesignBox: CollapseTensor added

5.14 GUIBox/JTextArea: the collapsed newfrat Tensor

5.15 DesignBox: Correspondence Analysis beans reconnected

5.16 GUIBox/JTextArea: the row scores of the collapsed newfrat Tensor

5.17 Properties for ExpandTensor: requiring Matrix Labels

5.18 DesignBox: ExpandTensor added
5.19 **GUIBox/JTextArea**: the row scores of the expanded `newfrat`.Tensor

5.20 **TensorToGraphArray** properties: Setting `includeWeights` to `true`

5.21 **TensorToGraphArray** properties: Setting `rowVertexLabel` to `actors`

5.22 DesignBox: **TensorToGraphArray** added

5.23 **GUIBox/JTextArea**: the **GraphArray** created by **TensorToGraphArray**

5.24 **GUIBox/MtSimpleFileChooser**: Changing the output filename for our Mage file

5.25 **GUIBox/JTextArea**: Output of `newfrat` data in Mage format

5.26 Mage: First view of the `newfrat` data

5.27 Mage: `newfrat` - 1st week with ties

5.28 Mage: animating the `newfrat` data

5.29 Mage: `newfrat` - all 15 weeks

5.30 **GraphArrayToMage** Properties: choosing a different edge weight cutoff

5.31 **GraphArrayToMage** Properties: choosing a different ball coloring method

5.32 Mage: `newfrat` - all 15 weeks (`T2GA.edgeWeightCutoff = 28.0`, `T2GA.ballColoringMethod = BY FRAME`)

5.33 **GUIBox/MtSimpleFileChooser**: Changing the output filename for our PDB file of Newcomb fraternity data

5.34 DesignBox: **TensorToGraphArray** (`T2GA`) reconnected to **GraphArrayToPDB** (`GA2PDB`)
5.35 GUIBox: Assuring the **RenderBasic** ChemicalDataFormat is set to PDB by hitting **Enter** again in the **JTextField**.

5.36 GUIBox: PDB output of Newcomb fraternity data.

5.37 GUIBox/JTextArea: PDB output of Newcomb fraternity data.

5.38 GUIBox/RenderBasic: PDB output of Newcomb fraternity data.

5.39 **GraphArrayToPDB** Properties Dialog: Changing the graph-ToExtract property to view last week of Newcomb data (NEWC15).

5.40 GUIBox/RenderBasic: Viewing matrix NEWC15 (16th week) of Newcomb fraternity data.

5.41 Mage: Viewing matrix NEWC15 of Newcomb fraternity data.

7.1 The Class Hierarchy of the Socnet Beans.

7.2 **ReWeight**’s relationship to **FunctionBean** and **StackLogger**.

7.3 **FunctionBean**’s basic data processing cycle.

7.4 Methods used in the **FunctionBean**’s data processing cycle.

7.5 Starting the code for **GraphArrayToMage**.

7.6 Declaring the fields for **ReverseTensor**.

7.7 Declaring the fields for **GraphArrayToMage**.

7.8 Programming the setter for the **InputGraphArray** property.

7.9 Programming the getter for the **InputGraphArray** input property.

7.10 Programming the accessor methods for the **OutputKinemage** output property.

7.11 Declaring the zero-parameter constructor for **GraphArrayToMage**.

7.12 Overriding **GraphArrayToMage.readyToCalculate()**.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.13</td>
<td>Overriding <strong>TensorToGraphArray.readyToCalculate()</strong></td>
</tr>
<tr>
<td>7.14</td>
<td>Overriding <strong>GraphArrayToMage.calculate()</strong></td>
</tr>
<tr>
<td>7.15</td>
<td>Overriding <strong>clearInputs()</strong> for <strong>GraphArrayToMage</strong></td>
</tr>
<tr>
<td>7.16</td>
<td>Turning off AutoCalculation and AutoClearing of Inputs for <strong>ReadDL</strong></td>
</tr>
<tr>
<td>7.17</td>
<td>Overriding <strong>setLogLevel()</strong> for <strong>GraphArrayToMage</strong></td>
</tr>
<tr>
<td>7.18</td>
<td>Running <strong>serialver</strong></td>
</tr>
<tr>
<td>7.19</td>
<td>Pasting in <strong>serialver</strong>'s output</td>
</tr>
<tr>
<td>7.20</td>
<td>Starting the code for <strong>BeanInfo</strong></td>
</tr>
<tr>
<td>7.21</td>
<td>Overriding <strong>getBeanDescriptor()</strong></td>
</tr>
<tr>
<td>7.22</td>
<td>Overriding <strong>getIcon()</strong></td>
</tr>
<tr>
<td>7.23</td>
<td>Overriding <strong>getPropertyDescriptors()</strong>: starting</td>
</tr>
<tr>
<td>7.24</td>
<td>Overriding <strong>getPropertyDescriptors()</strong>: declaring edited properties and their <strong>propertyEditors</strong></td>
</tr>
<tr>
<td>7.25</td>
<td>Overriding <strong>getPropertyDescriptors()</strong>: declaring already-existing properties</td>
</tr>
<tr>
<td>7.26</td>
<td>Overriding <strong>getPropertyDescriptors()</strong>: returning the array of properties to the caller</td>
</tr>
<tr>
<td>7.27</td>
<td>A snippet of the <strong>socnet_jastern.jar</strong> manifest</td>
</tr>
<tr>
<td>7.28</td>
<td>Saving your palette in Geovista Studio</td>
</tr>
<tr>
<td>7.29</td>
<td>Example of running the beans from Java code in a single-threaded environment</td>
</tr>
<tr>
<td>7.30</td>
<td>Simplified structure of a <strong>GraphArray</strong></td>
</tr>
</tbody>
</table>
# List of Tables

4.1 Srole Manhattan SES/Mental Health Data .................................. 86

6.1 Values for the LOGLEVEL Property ........................................ 221
6.2 Input Properties for **CollapseTensor** ..................................... 225
6.3 Callback Properties for **CollapseTensor** ................................. 225
6.4 Input Properties for **ExpandTensor** ...................................... 226
6.5 Callback Properties for **ExpandTensor** .................................... 227
6.6 Input Properties for **GraphArrayToMage (GA2M)** ...................... 228
6.7 Callback Properties for **GraphArrayToMage (GA2M)** .................. 229
6.8 Input Properties for **GraphArrayToPDB (GA2PDB)** ..................... 231
6.9 Callback Properties for **GraphArrayToPDB (GA2PDB)** ............... 232
6.10 Input Properties for **ObjectToString** ................................... 233
6.11 Callback Properties for **ObjectToString** ................................ 233
6.12 Input Properties for **OptScale** ........................................... 234
6.13 Callback Properties for **OptScale** ....................................... 234
6.14 Input Properties for **ReadDL** ............................................. 235
6.15 Callback Properties for **ReadDL** ......................................... 236
6.16 Input Properties for **ReverseTensor** .................................... 238
6.17 Callback Properties for **ReverseTensor** ................................ 238
6.18 Input Properties for **ReWeight** .......................................... 239
6.19 Callback Properties for **ReWeight** ....................................... 240
Acknowledgements

I would like to take this opportunity to thank some of the many people who have helped me with this project.

I thank Erika Bourguignon and James R. McLeod at Ohio State for their support and cultivation of a young undergraduate. I thank Dr McLeod for showing me what anthropology really could do, for his inspirational work, and mostly, for his friendship.

I thank Doug White for his encouragement and resolute support and protection in my early years as a graduate student. I thank him and Francois Heran for getting me to Paris, France where I met social networkers who were frustrated because of using software on various machines and operating systems – forming the start of this project. And I thank him for gracefully letting me go — even though neither of us wanted it — when it became necessary in order for me to proceed.

I thank Lin Freeman for his clear thinking and insistence on research focus and linear presentation. I thank him for his resolve to not let anyone into an orals/proposal presentation until a project is fully thought out and ready to implement. I thank him for protecting me from the bureaucracy, which he did without mention. I thank him for his patience in the long years it took me to get around to start getting some real work done. And I thank him for teaching me how to take ownership of my work.

I thank Kim Romney, John Boyd, and Sue Freeman for their patience and encouragement, as well as their support of a non-standard social science dissertation. I thank all four committee members for showing me how research was really done, how to have fun discovering social phenomena in a rigorous way, and
above all, how to be a colleague. Watching various combinations of them working
together on projects was one of the joys of my graduate education. This window
into their world was priceless.

I thank Tim Standish for his careful helpful and constructive comments on
the project proposal from a computer scientist’s point of view, and for his en-
couraging, supportive enthusiasm.

I thank Dao Vuong for his years of support and encouragement, and for his
good technical support, and keen philosophical, psychological, social, and political
mind. He has gotten me out of many a bind, and his many hours of insightful
consultation were worth their weight in gold.

I thank Brian Truong and all the other guys and gals in Social Sciences Com-
puting Services for putting up with me over the years while I work my way down
this path. I thank Brian Truong for his great help and technical support with
getting printers running in the final stages.

I thank Pat Skyhorse, Paula Trahan, and Tammy Chan for having the patience
to stick by me as friends and emotional confidantes through all my trials and
tribulations during this long journey.

I thank David Kongpiwatana Narong for re-familiarizing me with what hard
work looks like (and that we survive it), and for his prodding me to keep getting
back up on the horse.

I thank Bennet Tseng for his loving support through several of the hardest
years.

I thank Jerry Keys, David Leinen, and Kathy Alberti for their moral support.
These helped me get through some of the roughest times during the project’s life.
I thank them and the School of Social Sciences for the financial support — via
part-time work — I needed in the later years in order to complete this project.

I thank Angelo Ahn and Rose Delabella for returning me to a state of health and energy sufficient to allow me to pick up again and finish. This was no small feat on their part.

I thank Robert W. Keller for just being himself. Without Keller, this dissertation project would not have become a reality. I am grateful to have been in the right time and place to benefit from his knowledge and his love.

I thank my mother and father for valuing my education, and for all their financial and moral support over the long years. I could of course not have done it without them, and I appreciate all they gave.

I thank Sayon Syprasoeuth for being a loving partner who was ever patient with my erratic progress. I appreciate the space he gave me to finish it in the way I needed to, and the many hours of careful listening to my recounting the travails I encountered along the way. Without his loving support as well, this dissertation would not have been possible.

Finally, amidst an ocean of judgemental comments from people who did not know me, I am grateful for those kind comments from many folks who did their best to understand me whether they knew me well or not. Thom Batey once said he respected me for having the guts to “stick it out” long after others would have given up. Comments such as these were the life rafts that kept me afloat during the many years at sea.
Curriculum Vitae

Jeffrey Alan Stern

1963  Born, Lima, Ohio, USA.
1981  High School Diploma, Western Reserve Academy, Hudson, Ohio
1986  B.A., (Anthropology) (Summa Cum Laude, with Distinction), The Ohio State University
1986  Elected to Phi Beta Kappa
1987-1989  further study in Computer Science, The Ohio State University
1989  Chancellor’s Fellowship, University of California, Irvine
1989-1997  Teaching Assistant, Research Assistant, School of Social Sciences, University of California, Irvine
1994  Vice President, National Association of Student Anthropologists
1999  M.A., Social Science, University of California, Irvine
2003  Ph.D. In Social Science, University of California, Irvine
Social network research would benefit greatly if we changed some of the goals we have for our software. A software development framework fostering collaboration and cumulativity would make the greatest use of the limited financial and time resources available to our research community. The analyses we perform should not only be more easily shared, but also more easily visualized, taught, recorded, and openly developed.

Javabean architecture offers a place to start. In this dissertation I demonstrate some beans I’ve written for social network analysis — called Socnet beans — and their use in one graphical development environment called GeoVistaStudio. Socnet beans can log their activity, and have a standardized, thread-safe data processing cycle based on two parent classes written for this project — FunctionBean and StackLogger. These parent classes speed development of beans used in data flow designs — programs which run data through a series of transformations.

I also develop several data type support classes for the Socnet beans — classes
for the most common data types used by social network researchers. The two main classes in this group are **Tensor** — a 3-dimensional matrix, and **GraphArray** — an array of graphs. Future beans can be written more easily by other developers knowing that they have the Socnet bean parent classes and these data types available to them.

The dissertation includes a tutorial on the use of the beans for researchers using them in GeoVista Studio, as well as a guide for programmers wishing to develop more beans in a similar manner.
CHAPTER 1

Introduction

1.1 Background to the Problem

The idea for this project began when I was a teaching-assistant for an undergraduate class taught by Lin Freeman called, “Visualization of Social Networks”, here at the University of California Irvine. In this class Freeman was teaching students how to collect, enter, and analyze their own sociological data sets. These students collected data on their friendship networks, dormitories, church groups, athletic groups, and so on. After entering this data, they walked it through a series of transformations ending in a 3-D visualization of their data. I watched, overjoyed, as students rotated their data in 3-D space, and got insights into their own worlds. “Oh, yeah, he’s always been on the outside.” Or, “Wow, I didn’t realize she was in both groups.”

This class, more than any other I have witnessed, graphically demonstrated to students the reality of their social world — and the value of social network research.

Nevertheless, getting their data from a text file they entered to the final stage of visualization involved several steps. These steps were not so easy for the students to grasp – despite the fact that Freeman went slowly, over the course of weeks.
The process of teaching the students these transformations was so slow partly because they had so much to learn: both multi-dimensional scaling (MDS) and correspondence analysis (CA); the differences between binary and weighted or continuous data, and between 1-mode and 2-mode data; and how and why these were handled differently in the scaling transformations.

But I began to see another reason for the slow process, and it happened in the following way: Freeman had allowed me the privilege of jumping up every now and then to supplement his explanations with additional information, or to verify that the students had just understood something he had said. As we all approached the end of the course, I found myself more and more uncomfortable with the fact that the students just were not understanding why they had to learn all these seemingly “esoteric” steps. They seemed content to “jump through the hoops”, but they did not understand “why”. They just knew it led to an end visualization, and let it go at that.

One day, I found myself standing at the front of the computer lab, drawing a data-flow process on the white board. Here, I told them, is why you are doing this. Each program or step you are running your data through has a purpose: it either transforms the content of the data (the substance of the data themselves), or it transforms the format (the way the data are represented). Each step buys you something. Thus you are working with a flowchart, a pipeline of steps which gets you from start to finish. This did help some students, but for others, it seemed too little, too late.

Later, thinking about the entire process, I realized these students needed a single program or environment which linked all these steps together in a visual way emphasizing the data flow steps while they were working.

Thinking over the possibilities, I remembered taking a class led by Stephen
Franklin through the Computer Science department at UC Irvine a few years earlier. The class had been an introduction to 3-dimensional computer graphics, but had emphasized the use of a program called AVS, or Advanced Visualization System \cite{Advanced Visual Systems Inc. (2003)}.

AVS had precisely what the undergraduates in Freeman’s course needed: It allowed one to drag modules (small programs) down from a palette at the top, onto a worksheet, and connect them together in a network (or pipeline) of modules which formed a program. Starting from raw data, students and researchers could end up with beautiful 3-dimensional and even motion displays of their data. AVS was primarily used for weather and technology applications, but why not for social networks?

Thus, this dissertation project was born. The search for AVS revealed some problems with its use in social analysis, leading to IBM’s Open Data Explorer as an alternative, then on to programming my own specialized system in Java, and eventually to programming a set of functions based on a more standardized architecture – Java beans — and using them primarily, though not exclusively, in a graphical bean development environment called GeoVista Studio \cite{GeoVista Center (2003)}.

During this development process, more needs presented themselves. The complete list of these needs is presented below, and formed the design goals for this project.

### 1.2 The Problem

Social network research would benefit if the software which social network analysts used enabled them to do several things more easily:
1. Communicate, teach, and record visualization and analysis procedures which are often used in social network analysis.

For instance, when new students of social network analysis open a book on the subject such as [Wasserman and Faust (1994)], they find no practical tutorials for visualization methodology. [Weller and Romney (1990)] comes close in their discussion of MDS and Correspondence Analysis, but their tutorial is still not focused around the use of particular computer programs (and was not intended to be). Therefore, while these sources are excellent references for basic methodology, they are not specific enough to lead a student precisely through the same steps using the computer programs they will use.

Turning to the programs themselves, students encounter the same problem. Social network-analytic programs such as KrackPlot [Krackhardt et al. 2003], UCINET [Borgatti et al. 2003], or Pajek [Batagelj and Mrvar 1998, 2003] lack tutorials saying, “Start here: Here is how you take your data and turn it into a 2- or 3-dimensional graphic, or an animation.”

Thus, the books that are available talk on a more general, theoretical, or basic intro-to-methodology level, and the programs that are available do not present the user with an easy way into the logistics of the analytical process.

Furthermore, the architecture of our current programs is itself tailored for some needs but not others. For something “quick and dirty”, a menu-based program architecture offers a quick and easy way to get something done. But as a pedagogical tool, the menu interface falls flat: Simply presenting options, it hides the process, the flow. There is no easy way to discern which options are the most important and would apply to one’s data, or in what
A menu-based system is an obstacle also for operations on many similar datasets: there is often no easy way to set up “batch” (unattended and ordered) operations, and so the user has to run each data set through the set of transformation steps by hand repeatedly — a laborious error-prone process.

This process — the process of creating a visualization from social network data — involves many steps using our current set of tools. As stated above, these steps are presented in software only as unordered menu choices. Though unordered in software presentation, the researcher normally combines several of these in linear fashion into a data processing task in order to get a numeric or visual result. To date, I have never seen these linear orderings of choices communicated pragmatically in print. Yet social network analysis cannot be performed without this working knowledge, which is most often only communicated via word of mouth–either 1-on-1 in person, or in a rarely-taught class for the purpose.

Furthermore, no one piece of software contains all the steps involved in working with data. For instance, a simplified walk-through of data to create a viewable image might look like the following:

- enter your data into a text file format or in a spreadsheet
- import it into UCINET
- perform some form of metric scaling on it — an MDS or a CA, in order to get loadings — coordinates for viewing in 3-space (this may itself involve several pre- and post- transformative steps)
- export these coordinates back out again as plain text
• run some transformation program to get this ASCII file of coordinates converted into some form that an external chemical viewing program – such as Mage (Richardson and Richardson 2003) – can understand
• edit the Mage file, altering the color, radius, and labelling of the balls for optimal viewing and clarity
• open up with Mage and view

The above steps involve at least 3 separate programs, each with several sub-procedures. UCINET now comes packaged with Mage and Pajek, and UCINET now includes exports for each of these. However, each program still has to be started up separately, and imports and exports still have to be done manually by hand – though via menu choices now instead of via separate programs. Furthermore, other viewing data formats are not included as part of the package. For instance, if one wants to view the data in a different viewer, such as JaMM (Huffman and Bollinger 2000) or Moviemol (Ojamäe and Hermansson 2000), or as an image rendered by programs and browser plug-ins such as XMView (Huffman 2003), RasMol (Sayle 2003b), and Chime (Sayle 2003a) then often one will have to use a separate external conversion program to convert their data into a format these applications will understand, since the full set of exports is not built into UCINET (or any other program).1

2. Avoid the One Trick Pony problem

A major problem in small fields such as social network research is the one trick pony problem. This is when a researcher comes up with an algorithm helpful to the field, but cannot get other researchers or developers interested

---

1 One such set of converters comes in the form of separate BASIC programs for MS-DOS called the UCI2* suite of converters (Freeman 2001).
in including this algorithm in their programs (or at least not in a timely way). So the researcher writes his/her own program to support the new algorithm. But that is all the new application does.

Later, the researcher adds some minimal facilities to make the program a little more useful, but in the end, their field is littered with one trick pony programs that really do only a small subset of the operations necessary to do research.

This problem is actually a conglomeration of several other problems, some of which overlap with those listed below. For instance, research is slowed since the researcher has to take time out to program the entire application, in order that his or her algorithm may be supported.

As part of this process of writing an entirely new application, the researcher must usually develop his or her own internal data format to represent their data. Therefore the research must write additional import and export routines in order to get data into and out of the program and communicate with other programs. All of this represents duplicate effort, since ideally, the “network” of import/export routines would be “hub-and-spoke” for efficiency: everyone exporting to and importing from one common format.

Among one-trick-pony programs, this is rarely, if ever the case.

In the end, one trick ponies are problematic for the user, too. Their interfaces and command styles are not uniform across programs, thus requiring extra time to learn. Most cannot accept command-line parameters or deal with input and output streams, and so cannot be pipelined together.\footnote{This is less true of shell-based UNIX and Linux programs. FRAMES (\cite{Potmesil1987}) is a visualization environment which extends the UNIX piping philosophy. AL (\cite{Borgatti1985}), a social-network suite of programs, was MSDOS-based, and did not pipe streams, yet used the same philosophy of one executable program per process in a modular way, so as to build upon the expertise of separate authors.}
Therefore these programs require tedious and repetitive user-interaction.

3. Perform computational analyses and demonstrate them on any computer operating system, including mobile devices.

The one trick pony problem is also often — though not always — bound up with the next problem, the problem of non-multi-platform code, since many one trick pony programs have traditionally been written for a single operating system (or platform).

Social network researchers utilize a variety of computer platforms, from Macintosh to UNIX to MS-Windows and others. These researchers would be able to collaborate more productively if their operating systems, file formats and program availability were not obstacles. Furthermore, as machines are becoming increasingly portable, it would be nice if our analysis, entry, and visualization tools were available on these new devices, so that these activities could become more portable, as well.

4. Publish the latest structural analysis and layout algorithms.

Should a researcher want to expand upon one of today’s more popular social network programs by programming and adding in a new algorithm custom-tailored to his/her needs, it is difficult to do so.

The tools currently available for working with social network data each come with their own compiled-in versions of various transformation algorithms. These programs cannot be extended without the involvement of their authors.

For instance, a typical example is the set of spring embedding algorithms. There are many spring embedder variations, and no single package contains all of the more popular ones. And with the exception of Graphlet (Bran-
a package less commonly, if at all, used for social network analysis — none of the programs makes available in its menu choices more than three variations of spring embedder — a small subset.

5. Have separate, yet seamlessly integrated tools for data processing and visualization: Inter-operation and Loose coupling.

Most social network analysis programs come with their own packaged visualization engines, if at all. The one exception is now UCINET, which has just recently begun distribution with Pajek and Mage. UCINET also now comes with a network editor, as well. However, these programs do not provide output formats (export tools) for the wide range of other visual rendering languages.

Although packaging large programs together (as UCINET now does with Pajek and Mage) is a step in the right direction, there are still some distinct disadvantages, including:

(a) Having many parts of one program all under the responsibility of a single author means that some parts will fall behind in their development and/or support. This is especially true of programs including visualization engines together with algorithmic engines.

(b) If the programmer relies on a commercial, closed-source third-party module to support some of his/her code, then this module still has to be included in the program once during compile-time, and any new improvements to that third-party module will not be passed on to the user (via the end-program) until he or she purchases a new version of the whole program, or upgrades. This is a slow upgrade/improvement process.
Therefore, it would seem prudent to break down the functional parts of network-analytic software into finer parts which can be maintained and upgraded by many collaborating authors. These parts can be updated separately in an end-user’s program without affecting the other parts. Thus users could be assured of always having the latest version without having to wait for major updates.

(c) The user has to laboriously run his/her data through one program, then the next, exporting and importing data each time. A program with many parts could accomplish a great deal of flexibility in importing and exporting from and to various file formats. Any analysis engine would have to simply provide one output filter (export routine) for each type of viewer (VRML for Cosmo Player, a kinfile for Mage, etc.). A common data format would provide the lingua franca. Furthermore, an agreed-upon way of interacting between programs (in a way which requires that one program know nothing about another program except a standard interface — called loose coupling) would make a data pipeline feasible, and remove the need for manual import/export.

(d) If researchers experimenting with their own new algorithms wish to share these algorithms with other researchers, there is currently no mechanism for doing so easily and quickly.

6. Have access to many alternative programs

Currently, social network researchers do not have a wide variety of programs available to them for easily viewing (or otherwise processing) their data — at least not without going through the laborious process mentioned above. Furthermore, each program may have mutually exclusive advantages. For
instance, in the case of viewers, one viewing program may allow for better labelling of the viewed objects, while another program enables motion display. Neither does both, so the use of both programs would be helpful. If it were easy to get data into both formats for these programs simultaneously and view them side by side without a great deal of effort, the benefits of both programs could be achieved.

7. Batch processing

Related to the above problem is the batch-processing problem. Researchers often have more than one data set which must be moved through the same set of transformations. When the only way to do this is with a slow, tedious, and error-prone process of moving the data through one-trick-pony programs, research is hindered. If one could set up a pipeline one time, and have it run all datasets through, creating a set of output files (or images), research would be enabled.

8. Accommodate display of time-series data sets.

(Freeman, 2000) documents how in 1997 he had viewed email and listserv data he had collected together with Sue Freeman in 1980 (Freeman and Freeman, 1980). Seeing the data for the first time in motion, he was able to quickly identify four groups of actors based on their direction of motion, associating these groups clearly with each group’s actors’ opinions of the newly re-emerging social network field. These insights and others have led researchers in the field to acknowledge that for the field to progress, social network datasets will have to be increasingly dynamic (time-series-based), and tools for viewing them in motion will be required.

9. Accommodate large data sets
Tools built for the analysis of social networks have traditionally had severe limits on the size of the datasets they can handle. These limits have primarily been imposed by the compilers and operating system software – not the hardware. As memory and speed of computers have increased, our tools should be limited only by the memory and hard disk capacity (hardware) in future, so as to handle the larger data sets researchers now have.

10. Make their code available online and as part of other programs

More and more publications are being viewed online, if not published there as well. Already social network researchers have published articles with data sets requiring 3-D software to view them (Freeman 2000). Since the web is the future, it makes sense that programs – and even parts of them – be able to run online in a browser, or that our algorithms be able to be shared in web apps – web applications, or programs which run completely online via the person’s web browser. If algorithms are broken up into pieces of easily re-assemblable code, then programmers of web applications can use the same time-tested code used by researchers, by utilizing only parts of their programs – the modules containing specific algorithms.

11. Program their code easily

It would be helpful for those researchers who can code (and their assistants) if the coding of their algorithm were as easy as possible and involved as few as possible details relating to the architecture (hardware), platform (operating system), or environment (containing application). Thus, they could concentrate on the particulars of their algorithm and have to deal with a minimum of “housekeeping”, leading to more quickly developed, more numerous, and more robust implementations of a field’s algorithms.
12. Get new algorithms cheaply

Often researchers cannot pay for large expensive programs. It would be helpful if they could share algorithms and visualizers amongst themselves at a minimum expense.

13. Cooperate with researchers outside their field

Though it is nice to have software tailored to one’s specific field, it is better to have software which is also tailored to others’ fields, as well. Social networkers have been known to pull insights and algorithms from fields as diverse as psychology, biology, physics and electrical engineering. If researchers from these fields were to use the same program or software architecture, then facilities from these fields could be used, saving even more time (and leading to more insights).

1.3 Modular Visualization Environments as Potential Solution

1.3.1 Introduction to Modular Visualization Environments

At first, the easiest way I could see to solve some of the above problems was through the use of a modular visualization environment, or MVE, as I will refer to them here. Examples of commercial MVEs include AVS5 \cite{Advanced Visual Systems Inc 2003, Upson et al. 1989, Advanced Visual Systems Inc 1992}, Iris Explorer \cite{Numerical Algorithms Group Ltd 2002}, and Khoros \cite{Khoral Inc}. Programs in this genre do not have a single category name to describe them (such as “word processor” or “spreadsheet program”). They have been variously called “application builders”, “data-flow network applications”, “data-flow systems”, and even just “scientific visualization systems”, since many such systems incorporate a data-flow paradigm anyway. I favor the use of modular visualization environment or just MVE, the term adopted by \cite{Wood 1998}, since it seems to best capture both the architecture and purpose of such systems. 
MVEs allow users to visually create programs which are really data-flow pipelines\footnote{Also, the graphical networks that these systems create have been called everything from “map” (Iris) to “visual program” (OpenDX and Khoros), to “network” (AVS) and pipeline (ApE) \cite{Song1994} and even “executable function network”. I will normally use \textit{data-flow pipeline} to refer to the network, or pipeline of functioning modules such a system creates.} for running their data through the various steps in the visualization process. Each program-as-pipeline consists of a string of these steps, realized as \textit{modules}, and the modules are strung together by the user. This process closely follows the process a researcher normally would put their data through anyway, the modules matching each major step or transformation that their data must undergo.

To get a brief idea of what an MVE session looks like, consider Figure \ref{fig:MVE_overview}, a screen-shot taken from an online IBM tutorial for using OpenDX \cite{LloydA.Treinish2001}. In this picture may be seen OpenDX’s “Visual Program Editor” window. The user is in the process of creating a view of the two hemispheres of the earth based upon the ozone data as “pseudo-colored deformed surfaces”. The user does no text-based programming. Each module represents a data transformation. From the top, where temperature and ozone data are imported from a data file (controlled by a Sequencer which steps the Import module through reading the data pieces at a time), to the near-bottom, where the data are composed together with world-map projections, and rendered in an Image window, the program
Figure 1.1: OpenDX

comprises a data-flow pipeline.

Notice that the outside window (the “Visual Program Editor”) has two parts. The left hand side contains two lists. The top list categorizes the modules available in its “Categories” section. Currently the “Rendering” category is selected. Below the categories is another list of modules available in that category. The right-hand side is called the “canvas”, where the data-flow diagram (which has been moved to the left-most part of the canvas) may be seen. There is another window sitting on top of the Visual Program Editor window. This window, called the “Image” window, belongs to the “Image” module close to the bottom of the data-flow pipeline. Finally there is a control window for the Sequencer module, allowing the user to control the motion of the visualization by controlling the
output signals of the Sequencer.

The creation (or “visual programming”) of such a data-flow pipeline is essentially a three-step process, using the computer’s pointing and selection device, or mouse. First, the user drags (or clicks) modules from the library list onto the network palette, canvas, or network editor. In this case, the user has clicked several modules out onto the canvas (including the “Image” module from the “Rendering” list). Second, the user connects modules together. This is done by “drawing” lines which connect an output “port” on the edge of one module with an input port on another module. Third, when all modules are in place and all required ports are connected, the program (the data-flow pipeline) is complete and ready to run, and the user clicks the “Execute” menu choice. The picture seen in Figure 1.1 shows the results (in the separate “Image” window) of the data-flow pipeline in mid-run: the user has executed the data-flow pipeline and one of the frames of the motion video is currently being shown in the Image window.

1.3.2 Architecture and Advantages of MVEs

Modular Visualization Environments began to emerge in the late 1980’s and early 1990’s as it became apparent that this type of architecture was both needed by and advantageous to the scientific research community.

MVEs are based upon an entirely different type of software architecture. Such an architecture separates out the operating, organizing and maintenance facilities of the running program from the program’s capabilities. These capabilities are themselves distributed as separate modules with a standard agreed-upon way of communicating between them. This modularized architecture is what Song (1994, pp. 6-7) refers to as a “reductionist” model with “loose coupling”, as

16
opposed to a “holistic” model where the application is compiled together into one monolithic executable. Wood (1998) calls these “turnkey” systems, and “modular visualization environments” (MVEs), respectively. This design allowed modules to be strung together in dataflow pipelines. These pipelines themselves may be saved in their design form, and alternatively may be actually compiled into fully functional stand-alone user programs. Song (1994, chap. 2) reviews a more complete list of MVEs from prototype and academic to commercial, analyzing their various strengths from the point of view of his dissertation involving memory management and speed under such architectures. Out of pragmatism, I consider here only those which are in widespread use and have a large user base, whether commercial or shareware. I focus less on one-time academic experiments and prototypes, though much of the following applies to them as well.

MVEs are good news for researchers in many fields for the following reasons:

1. MVEs are extremely general tools

   Because of their architecture, the number and types of visualizations that MVEs can muster are virtually limitless. One can gain an idea of the number and variety of visualizations of which AVS and OpenDX are capable of by visiting the AVS home page on the World Wide Web (Advanced Visual Systems Inc, 2003) as well as the AVS international user community’s online collection of customer success stories (International AVS Center, 2003), and the OpenDX web site (OpenDX Open Source Software Project, 2003).

2. MVEs are modularized

   As mentioned earlier, an MVE separates out the capabilities of the program from the core maintenance procedures. These “capabilities” are really just various algorithms for data analysis, which can be treated as filters on data.
These filters are called “modules”. Modules can be plugged together into data pipelines, or programs, in almost any combination, as long as the data-types passed between modules match.

3. MVEs are expandable

Usually an MVE comes with a large set of modules, out of the box. But if the user does not find a module that will do what he or she needs, then the user may create a module that does. The collection of AVS modules is thus ever-growing.

4. MVEs are visual

MVEs allow visual programming at its easiest. The data-flow visualization architecture has enjoyed such popularity with researchers in many fields because its architecture, appearance, and graphical user interface intuitively match their work flow.

5. MVE programming is fast

Finally, and most important, creating programs with MVE modules is fast. This point should not be missed, since really it is the final goal of this type of architecture. When all the hoopla has passed, what it really comes down to is that researchers can assemble “programs” to pipeline their data into a meaningful, explorable visual presentation, in a fraction of the time required using other menu-oriented programs, or suites of programs. Furthermore, MVE network programs (dataflow pipelines) can handle batch-processing of long jobs.
1.3.3 Dataflow Architectures: Specific Advantages to Social Network Researchers

How do the generalized advantages of MVE modularized architecture address the problems peculiar to social network analytic software? The following is a list of what MVEs have to offer:

1. Communication and teaching (Problem 1)

   Plugging modules together in a graphical environment, a user can easily see how they are walking data through a pipeline of various filters to obtain a 3-dimensional image.

2. Avoid the one trick pony problem (problem 2)

   A second primary strength of module-based design is that researchers may combine modules in a single environment. This means they do not have to re-program a one trick pony. Instead, an MVE is their application (or, if the stand-alone application their MVE builds is separate, then it may still contain tools combined in from other authors). The data-flow pipeline simply consists of the researcher’s programmed algorithm (in form of a module) along with other modules from other researchers.

   Furthermore, export and import methods are collected into a single framework for efficiency and coordination.

   Finally, since these networks/pipelines are all created inside of the MVE, the MVE itself provides the consistent interface. The user only has to learn to use their MVE. Modules are presented by the MVE in a uniform manner.

3. Cross-platform access (problem 3)
Most MVEs are available on a limited number of platforms. While limited, this is highly superior to the single-platform alternatives currently available to social networkers. The platforms were originally only the more popular UNIX variants, such as Sun’s Solaris, SGI’s Irix, and so on, but now include Windows, Linux, and Macintosh (OS-X).

4. Publication, review, and a faster research and development cycle (problem 4)

A modularized system allows for plugging in of new modules, meaning that researchers may share their expertise and experiment with new methods amongst themselves, without having to wait for developers to include it in their monolithic software. This can only speed the process of research.

A researcher can also choose to offer a module as open-source, for review and collaboration with peers.

5. Inter-operation and loose coupling (problem 5)

Having code in modules also helps with the problem of interoperation. The modules are integrated, working all in one environment seemlessly. Yet the modules, while working more closely together than separate standalone programs, nevertheless do not depend upon the development of the other modules in order to function. Thus they are somewhat loosely coupled.

6. Batch processing (problem 7)

Certain types of MVE modules, (OpenDX calls theirs sequencers), can be used to trigger the network to run several times. Together with a list of input data sources, this meant that the MVE can run stand-alone, and thus process batch jobs. In such a case long jobs can be processed without user-intervention.
7. Large and Time-sequenced data sets (problems 8 and 9)

MVEs are also attractive because they can handle both large and time-series data-sets easily (with some restrictions on large data-sets — c.f. (Song 1994)).

8. Easy to Code (problem 11)

MVEs offer an alternative to one trick pony applications. The programmer can focus on programming their algorithm (as a module) and not have to worry about creating the entire application.


OpenDX is free but AVS, Iris Explorer, and Khoros are not. So here OpenDX has a potential advantage over the others as an environment scientists could build on.

10. Outside-field cooperation (problem 13)

MVEs are used by researchers in many fields, including meteorology, engineering, and chemistry. Having their algorithms immediately available as modules in the same program would be a bonus to social network researchers.

1.3.4 Disadvantages of MVEs

While these programs seemed to be good first cuts at the problems listed above, none of them were able to adequately address all of the issues listed. Some problems remained:

1. Cross-platform access (problem 3)
While most current MVEs are more cross-platform than the single-platform programs available to social network researchers, they are still limited in several ways. These programs are constructed using the C programming language, which, despite ANSI standards, still varies from among compilers and hardware and O/S architectures. This means that some manual translation of the source code – called “porting” – is needed each time the designers wish another operating system to be able to execute the programs. This porting activity is labor intensive, often involving a separate staff for each of the new ports (and accompanying documentation). Thus, every time a market develops on another operating system, the potentially expensive decision to add a port to the number the company already supports is not taken lightly. One of the factors, of course, is whether a particular operating system’s market is “profitable”. And since the types of superior operating systems that researchers often prefer are not popular with the mass market, even a technically oriented corporation such as AVS can find itself excluding some researchers.

With OpenDX for example, this is not quite as much an encumbrance, because OpenDX is now completely open-sourced. Yet it still involves bringing together a separate team interested in porting to the operating system. For instance, there are several ports of the Linux version, in various states of disarray, all requiring bug-fixes, etc. Their ease of installation varies, and bugs relating to specific platforms mean that actual support for those bugs is in reality limited to a few support team members who are familiar with that particular platform’s port of the product.

2. Loose-coupling (problem 5)

While the modules in MVEs interoperate beautifully in their respective
environments, they are still not completely loosely coupled. They are still tied to their specific systems and the library calls for that system. An AVS module cannot work in Iris Explorer or visa versa, and so on.

3. Alternatives for researchers (problem [6])

While having healthy choices between “competing” modules is possible in these MVEs, these alternatives are still limited. It would be nice for each program to have the ability to use each other’s modules, or modules from other programs compiled in the same language. To this end, the user does not have as many alternatives available to him/her as might be possible with a system which could use modules from other programs.

4. Alternatives for programmers (problem [10])

It is not possible to use modules from most of today’s MVEs as they are, in a web application, or in any other application of the same language. The programmer of other programs (web or otherwise) cannot take advantage of the modules made for an MVE by simply including them in his program. He or she is forced to use their MVE to create the entire program. Furthermore, no MVE to date can as yet create programs which run on the web, within a browser window on any system, without special software.

5. Ease of coding (problem [11])

While creating a module for an MVE is shorter than coding an entire application, there is still a degree of difficulty involved with familiarizing one’s self with the library calls that must be made by a module in these environments. This “overhead” would preferably be minimized, so that the author could concentrate on the business that the module engages in (the algorithm, for instance).
6. Cost (problem 12)

Although OpenDX is free, AVS, Iris Explorer, and Khoros are quite expensive. This limits their accessibility so their viability as the basis for an environment which would maximize cooperation among researchers. In fact, when I had first encountered AVS, it was through my university’s computer science department, which had access to it solely on contingency that a class be taught with it: the department itself could not otherwise afford its purchase. After the classes were taught, the software was removed.

1.4 A Cross-platform Rewrite

On one hand AVS, OpenDX, and Iris Explorer, by virtue of their architecture, could solve many of the current problems which our current software have not, or could not. On the other hand, because of their various limitations, they also did not appear to have potential for solving many of our future needs.

I therefore looked for an MVE written in an extremely cross-platform language such as Java, Tcl-Tk, perl, python, and so on. The only one I was able to find was Sieve (Sieve Project 1997), a small experimental prototype MVE written in Java by researchers at the Virginia Tech Department of Computer Science. Sieve specialized in allowing two researchers on different machines to view a data-flow pipeline at the same time — a novel idea and definitely one in the interests of collaborative research and teaching. But work on Sieve had been discontinued, and its experimental state made it a less than realistic alternative.

I decided at this point to think about writing my own MVE in a cross-platform language. Such a system would solve the remaining problems listed above: it would be maximally cross-platform; its modules would be completely uncoupled
so that they could be developed independently; it would be easy to code, removing much of the overhead of managing the modules; and it would be free so as to encourage maximal usage and cooperation in the field. The modules would be open-sourced, and thus easy for academics to access and review. The system would provide a healthy environment for research collaboration.

I started with Java, because of its similarity to C, the care Sun had taken in its design, and its ubiquitousness. However, as I started into learning about how to set up the architecture for such a system, I found it a formidable task. There was a considerable amount of overhead involved in managing the timing and flow of data from module to module. Furthermore, modules themselves had to be thread-safe. That is, they had to be able to be accessed simultaneously by more than one other component\footnote{More precisely, they should have more than one program-counter running through them at the same time. This is also called re-entrant code.} This led me to experiment with some developmental modules based upon Paul Hyde’s CubbyHole and BooleanLock Java classes\cite{Hyde1999}. While I got some multi-threaded modules going, I still needed to build the environment to run them in. How to manage the interaction of the modules?

I started looking for information on the theory of how AVS and OpenDX were programmed. However, there was very little in the literature on modular visualization environments. AVS’s primary article had only introductory material describing its actual architecture\cite{Upson1989}. However, it did give one clue to how AVS managed to control the modules. AVS had what they called a flow executive, “..the component of the system that determines when to run various modules. It is implemented much like an operating system scheduler.”\cite{Upson1989} Writing such a system was something I was beginning to have doubts about being able to complete in a timely way for a dissertation.
The AVS Developer’s Guide (Advanced Visual Systems Inc. 1992) concentrated on the kind of help a programmer of an AVS module would need, but (probably for obvious reasons) avoided discussion of the AVS environment’s own internal architecture, much less details about its flow executive. There were many articles available on both OpenDX and AVS related to case examples of their use, none helpful to my cause. Years earlier, researchers at the Ohio Supercomputer Graphics Project at The Ohio State University had developed a prototype of a modular visualization environment, called “apE” (Dyer 1990), but the rights to apE were sold to a European company and the development team has since disbanded.

I had to tackle the issue of exactly how to get data from one module to the next in such a system. Would I use Java’s serialization method, where any properly-configured object may be turned into a stream of bits and sent somewhere else for storage and later retrieval? Or would I use Java piped streams\(^7\) to effect the exchange? Experimenting with these for a while, I noticed that while it was possible to use these, they offered no support for synchronous data communication. More specifically, they were not designed to control data coming in cycles, which would mean more coding on my part.

I spent quite a while experimenting with and developing Java classes for “ports” on modules and working out a control protocol for the transmission of data. But then each module had to have an input port controller and output port controller, monitoring their respective arrays of ports. Furthermore, I still had to make decisions about how the entire network would run. Would it be push or pull? That is, would modules re-execute based upon data coming in to them from upstream modules (push)? Or would they execute only as changes below them

\(^6\)Pete Carswell, Ohio Supercomputer Center, communication via electronic mail

\(^7\)java.io.PipedInputStream and java.io.PipedOutputStream
occurred, re-using data from a previous execution (pull)? \[\text{Upson et al. (1989, p.36)}\] had mentioned this as a design choice.

I also spent considerable time working out designs for 3-dimensional matrix and graph Java classes.

It was getting complex, at all levels of the job. And a bit too large.

1.5 Java beans to the rescue

Finally, it was suggested to me\(^8\) that I take another look at Sun’s JavaBeans component architecture (\[\text{Sun Microsystems Inc. (2003c)}\]). JavaBeans are a way of packaging up pieces of code in a way which can later be more quickly incorporated into other software. Thus, it handled problem \([10]\) allowing programmers to share their code more easily.

JavaBeans had some other nice advantages. They:

- are not necessarily proprietary
- are based on a free community-based cross-platform language
- are very loosely coupled
- are highly flexible
- may be used in a flow-based environment, but may also be used anywhere else
  - to create drop-down menu items
  - to create pieces of other programs

\(^8\)Danyel Fisher, personal communication
– even can be used as pieces of web applications or web services on web sites

* Java programs themselves could be run in a browser (applet) or as a stand-alone program (application)

But I quickly found that their very flexibility also made Java beans somewhat problematic.

1. **Java beans not restricted to, nor designed for a dataflow type system**

   The same flexibility that gives JavaBeans the advantage of generality also hinders their use in a specialized system: Since they are not tailored to specific uses such as being a function in an data-flow pipeline, they will not automatically solve the problems associated in such an environment. Beans for instance are not set up to operate according to a data flow processing “cycle” – which modules in an data-flow pipeline must.

   Furthermore, JavaBeans do not have “ports” for input and output. They simply have procedures that may be called.

2. **Java beans have housekeeping overhead**

   Java beans can seem easy to work with at first, but one quickly realizes that potentially, there is a lot to manage with a bean. Issues such as synchronization and multi-threading, association with icons in a visual environment, and so on, are extraneous to the object of encapsulating the function of a module (its algorithm). These are housekeeping issues which I had hoped to insulate the writer of modules from.

---

\textsuperscript{9} getters and setters, c.f. Section 3.4.1
3. There was still no flow-based environment (like AVS or OpenDX) for them, so I would have to write my own

1.6 GeoVista Studio

Fortunately, a group of researchers at the GeoVista Center in the Department of Geography at the University of Pennsylvania had stumbled across this same family of problems. These researchers were part of a subfield of geography called geoscientific data analysis and visualization. Thoughtful consideration of software infrastructure, they had decided, was an important issue that researchers in their field had preferred not to focus on (Takatsuka and Gahegan, 2002, p.1133). Setting up a project and funding for it, they too, had seen the benefits of data-flow-based environments such as AVS and OpenDX. For similar reasons, they decided to devote a full-time programmer (Takatsuka) as well as several graduate student programmers to the aim of writing their own program from the ground up in Java, called GeoVista Studio (GeoVista Center, 2003), as well as geoscientific JavaBeans that would run within it (or any other JavaBean-specific environment).

GeoVista Studio (or just Studio) is not a modular visualization environment in the strict sense because its modules are Java beans, not proprietary modules which work only within the data-flow system. Furthermore, there is more than one way to connect Java beans together in the system. Yet Studio does present the layout of the connection of the beans in a manner equivalent (to the user) to a data-flow pipeline: It emphasizes the flow of data through the Java beans. See Figure 4.61 for a view of Studio’s “Design Box” window, containing a full-functioning data-flow pipeline for 2-mode data analysis.
Because of its creative design, as well as its foundation on Java and Java beans, Studio addresses several of the problems listed above which traditional modular visualization environments could not:

1. Cross-platform access (problem 3)

   Since Studio is itself programmed in Java, and builds programs which run in Java. Java is extremely cross-platform. Thus there is only one copy of the code to maintain and support. There are no separate versions to recompile and port to different systems, or support separately. This is the basis for Sun’s motto for Java, “Write once, run anywhere.”

2. Loose coupling (problem 5)

   Studio requires nothing of the beans except that they conform to the JavaBeans coding standards (Sun Microsystems Inc [1997]). There is nothing additional a programmer of a JavaBean must do to make it work within GeoVista Studio. This is maximally loose-coupling. JavaBeans can work outside of GeoVista Studio (in any other program or Bean-friendly environment) as well.

   Furthermore, interoperation with other JavaBeans within Studio is seamless, because Studio handles the various methods that JavaBeans have for intercommunication, presenting these various methods to the user (visual programmer) through a consistent, uniform interface.

3. Alternatives for Researchers (problem 6) and Programmers (problem 10)

   Although GeoVista Studio is a programming environment, it may also be used by researchers as a research (run-time) environment. This is due to

\[\text{c.f. Figure 3.4}\]
the nature of Java, and to GeoVista Studio, which allow beans to be loaded and running within Studio before necessarily compiling the entire program together into a monolithic application.

Therefore, users (researchers, who are building programs which appear visually in Studio as data-flow pipelines) have many options open to them, since they may pull beans in from many different sources, include them in GeoVista Studio, and have use of them without any additional code (text) being added. This includes both algorithms and visualization engines, or renderers\footnote{Examples are given with one such outside set of JavaBeans in Sections 4.6 and 5.11}.

Furthermore, programmers may use the JavaBeans they wrote to work in Studio in other environments as well.

4. Ease of Coding (problem \[11\])

While JavaBeans can seem complicated at first for a programmer, and there can be a lot to manage, they are still simpler to code than an AVS or OpenDX module. And since JavaBeans are a more widely-used coding device, there are online tutorials such as (Quinn, 2003) and books such as (Englander, 1997), (Doherty and Leinecker, 2000), and many others (Sun Microsystems Inc, 2003f).

5. Cost (problem \[12\])

GeoVista Studio itself is free, though the JavaBeans used in it may be either free or commercially-obtained. This makes Studio a maximally-accessible environment for researchers operating on a budget.

6. Code re-use (problem \[10\])
Java beans are not limited to use just in Studio. The same beans may be used in many other non-data-flow programming environments as well, such as such as Borland’s JBuilder (Borland Software Corporation 2003), Sun’s Sun One Studio Sun Microsystems Inc (2003i), and many others. The beans will, of course, also run in the testing environments such as Sun’s Bean Builder (Sun Microsystems Inc 2003a), and the Bean Box application which comes with Sun’s Beans Developer Kit (Sun Microsystems Inc 2003g). They can also be incorporated into Java servlets (Sun Microsystems Inc 2003d) and JavaServer Pages (JSP’s) (Sun Microsystems Inc 2003h) as part of dynamic web pages and web services (Sun Microsystems Inc 2003j).

These advantages left Studio as the final candidate for non-programming researchers to use JavaBeans with, in a manner emphasizing data flow.

1.7 Problems with JavaBeans

It might seem at this point that I was ready to go. However, JavaBeans still had a major problem which inhibited some of the design goals set out for this project. This problem consisted in their generality. JavaBeans were not designed specifically for the task of representing modules in a data-flow environment. They can work for this purpose, but they were designed to be much more general. A JavaBean can be virtually anything; a text area to type into, a slider to rate your satisfaction with a movie, a calculation to raise the temperature of a tank of water – anything. JavaBeans may be either visual, in which case they represent themselves on screen in some way, or non-visual, working in the background.

This generality gives JavaBeans an amazing amount of flexibility, but this flexibility has three major downsides for their use in a data-flow environment.
These three downsides have implications for the user wanting to utilize JavaBeans as modules in a data-flow research environment.

1. The method for connecting JavaBeans is unstandardized

   The first of these downsides is that the JavaBean specification does not include any normative way to connect JavaBeans together to transfer data. Therefore, programmers of JavaBeans have worked out their own solutions for doing so. This has resulted in several different schemas for connecting JavaBeans together, but no single standardized way. This makes JavaBeans more difficult to use for end-users — contrary to project design goal.

   Fortunately, the programmer of Geovista Studio, Masahiro Takatsuka, has done a nice job of evaluating and incorporating into Studio all the most popular methods of connecting beans together. Furthermore, he has programmed Studio so that it presents these to the user in a helpful “Adapter Wizard”. This means Studio can accommodate most any JavaBean and connect it with any other, all in a visual way. This is a tremendous step forward in making JavaBeans everywhere more useful.

   Nevertheless, for the user, this is still an extra step they may not understand. After all, choosing among connection methods is a task completely extraneous to the research at hand.

2. JavaBeans have no inherent internal data-processing cycle

   The second downside to this generality is that JavaBeans have no inherent data-processing cycle. The core of a data-processing cycle is timing. There

---

For instance, Figure 4.9 shows the Adapter Wizard. Currently the method of connecting the two beans is “using Source’s Getter”. Clicking on the the “Selector” button above it will drop down other choices.
is a period during which the module waits for all inputs\textsuperscript{13}. The module
does not begin computation until this part of the cycle is over. Then the
module would have a time when it began execution, executed its task, and
ended. Then it would have a time when the output or outputs from that
execution were offered up to other beans or to the user, and finally a time
when that data transfer actually occurred.

None of this is directed by the JavaBean specification. And it makes sense
that it would not. A data-processing cycle might make sense to a data-flow
module, but it would not make sense necessarily to a ratings slider bean or a
text window bean. Therefore, the generality of the JavaBeans specification
does not include a data-processing cycle.

Without such a synchronous data-processing cycle, JavaBeans simply cannot
work in a flow environment. Therefore, such a cycle has to be pro-
grammed into the JavaBean itself. However, doing so requires extra pro-
gramming for the programmer of the JavaBean. This is contrary to design
gold 11 -- to make the coding of modules as easy as possible so that the coder
could concentrate on his/her algorithm, not on the "housekeeping." The
more housekeeping a programmer has to learn and manage, the more error-
prone are the modules, and the fewer modules end up being programmed,
because programmers search for other solutions.

3. No built-in error and activity logging

The third downside to the generality of the JavaBeans also related indi-
directly to the problem of ease of use and interoperability (design goal 5).
This problem became apparent to me only later, during programming.

Error-reporting is an often-overlooked part of programming for many (if

\textsuperscript{13}or operands in traditional computer science data-flow processing terminology
not most) applications. This may in part be owed to the architecture of many GUI programs, which are today menu-driven. It may be more apparent in a menu-driven environment where a bug is, or where user-error may have taken place. However, in a complicated data-flow environment, many modules are functioning. When there is an error and the data-flow pipeline stops, how can the user tell where the error occurred? Furthermore, how may they discern the level of the error, such as whether the problem is of their doing, or the programmer’s? This could be quite frustrating for a user. Thus the ability to have modules log their activity became an important design sub-goal, despite the fact that JavaBeans did not inherently do so.

1.8 JavaBeans extended: Ancestor classes

Geovista Studio went a long way to making beans work in a data-flow visual environment. Indeed, without Studio — or something like it — this project would not have been possible within the limits of a doctoral dissertation. Yet if I were going to make progress towards the design goals listed above, I would have to go further in addressing the above three issues.

For this project, I therefore created a class of Java object called FunctionBean, which itself descends from (i.e., inherits all the functionality of\textsuperscript{14}) another class of object, called StackLogger. These two classes — as well as other support classes — were written specifically for this project. They are called ancestor or parent classes, because all Socnet beans descend from these classes. Therefore, all Socnet beans are also FunctionBeans and StackLoggers, and inherit all their functionality. This descendence is crucial to the functionality of the Socnet beans and makes them much easier to use than normal Javabees — both for

\textsuperscript{14}Chapter 3 reviews Java, Java beans, and basic object oriented programming terminology
Studio users and for the beans’ programmers.

**StackLogger** logs all activity of the bean, and **FunctionBean** incorporates a built-in data-processing cycle, allowing a programmer to turn a bean into a function (or module) in a data-flow pipeline. Handling of a uniform interface for interconnection simply involved the addition of a guideline for use, and a simple extension to the JavaBean naming convention. These three together form a powerful way to more easily write modules for data-flow pipelines in Studio, which will nonetheless still perform as normal JavaBeans outside the Studio environment. Furthermore, Studio itself did not have to be altered to accommodate beans based on **FunctionBean** and **StackLogger**.

Explanation of the **FunctionBean** and **StackLogger**, as well as the interconnection guideline and naming convention, will be covered in Chapter 6.

1.9 Data types

1.9.1 Introduction

However, there was still one problem yet to be solved: data. In order for beans to connect easily, not only should they have a consistent link-up mechanism; also they should have agreed-upon data types with which to inter-communicate. That is, if one bean’s output is a matrix, but the Java class that holds this matrix is not recognized by the next (downstream) bean, then these two beans cannot work together, and a data-flow pipeline cannot be built.

The most common general data processing data types used in social network analysis are twofold. The first is a data format which can hold either 1-mode (such as attribute-by-attribute or actor-by-actor) or 2-mode (e.g., object-by-attribute, or actor-by-event) data in some form, such as a table, or matrix (though there
are other methods of holding the same data, such as adjacency lists, blocks, and so on). The second form is a visual format to represent graphs and other objects in 2- or 3- dimensional space and possibly also represent their connections, as edges or arcs. Other data types are possible (and needed) but from the point of view of the project, these would be the place to start in supporting social network research data. Yet Studio came packaged with no 3-d matrix or graph data types.

1.9.2 Tensor and GraphArray

Therefore, for this project, I created two Java data types which could be passed between modules. The first is called socnet.core.Tensor, or just Tensor, which represents a 3-dimensional matrix.

socnet.Tensor is built upon some other support classes, some built specifically for this project, but not listed here, and upon third-party objects, such as Jama.Matrix class, which represents a 2-dimensional matrix. Jama.Matrix is part of JAMA [National Institute of Standards and Technology and The MathWorks, 2003], a basic linear algebra package for Java. JAMA includes the ability to reliably calculate a Singular Valued Decomposition on a matrix. SVD is a core part of the correspondence analysis (CA) realized by the beans programmed for this project.

I also created a general class for handling graphs called socnet.core.GraphArray, or just GraphArray for short. GraphArray is a class of object which can represent a set of graphs such as over a range of time. The graphs may or may not include edges. The class does not include drawing routines, simply storage and representation in as general a way as I could see possible. Furthermore, GraphArray can keep track of arbitrary subsets of objects (e.g. actors) or connections (e.g., their relations) within the full set, so that they may be dif-
ferentiated visually (e.g., by color, shape, etc.) in the final screen representation.

As with Tensor, GraphArray is built upon work by others, in this case the Data Structures Library in Java (JDSL Team 2003). I made use of the JDSL Group’s JDSL.IncidenceListGraph for the basic graph and the JDSL.Red-BlackTree to keep track of the nodes in the socnet.core.GroupedTree.

GraphArray also depends on some support classes I wrote in order to be able to iterate through the groups in a GraphArray: AttribGroup, GroupIterator, GroupTree, and GroupedGraph. These classes allow storage, retrieval, and navigation through the subsets of vertices and edges in a GraphArray.

Like FunctionBean, Tensors and GraphArrays are also StackLoggers, so they can log their activities and return error codes in a graceful way, when being used inside a Java bean.

1.9.3 Other Data Types

However — unlike with AVS and Open Data Explorer — using JavaBeans as the basis for modules means that data types are completely open. It is not necessary to develop beans which must use Tensor and GraphArray. If one likes, and as the field progresses, one may find or develop alternative or superior data types, and write FunctionBeans around these new data types. Older FunctionBeans are not obviated by newer data types. Instead, all that may be needed in such a case is to write two FunctionBeans to translate back and forth between the old and new data types.
1.10 Conclusion

With StackLogger, FunctionBean, Tensor, GraphArray, and their support classes, beans could now be written to import from a text file into a Tensor or GraphArray, to convert from a Tensor to a GraphArray, to manipulate Tensors and GraphArrays, and to export to various formats.

The first of these FunctionBean data-flow beans are what I created for this dissertation project, and are called the Socnet beans. There are fifteen Socnet beans — enough to do simple analyses of 1-mode and 2-mode datasets, producing two equivalent visual images of each dataset.

Chapter 2 contains instructions for installation of Java, GeoVista Studio, and the Socnet beans. A pretutorial for understanding Java, Java bean and object-oriented terminology used throughout this dissertation is the subject of chapter 3. Chapters 4 and 5 are tutorials for creating visualizations of 2- and 1-mode datasets. Chapter 6 is a user’s guide where basic reference information on the data types, ancestor classes, and beans created for the project are covered. Chapter 7 is a programmer’s guide and reference for researchers wishing to write their own data-flow beans based on StackLogger and FunctionBean. The dissertation concludes with chapter 8 summarizing the content of the project and making some observations on possible future work.
CHAPTER 2

Installation

Before the beans can be installed in a visual bean-building environment, you will need to have certain software up and running. This section will cover installation of the socnet beans, as well as their required software.

Though the beans may be used in any environment, I demonstrate set-up for GeoVista Studio in particular. While beans may be used simply by programmers who type in their code, they can also be used in many bean-aware graphical integrated development environments (IDEs), such as Sun ONE Studio (formerly Forte/NetBeans), WebGain VisualCafe, Borland JBuilder, IBM Visual Age for Java and the like. However, to date, there are only two major development environments which are purely visual, which is to say, requiring no coding at all. These are Sun’s “BeanBuilder” environment, and Penn State University’s GeoVista Studio. Of these two, only Studio emphasizes flow of data from one bean to the next, and for this reason, Studio maximizes the use of beans in a collaborative environment. Setup for Studio is described in this section.

1http://www.geovistastudio.psu.edu
2Since this dissertation was started, GeoVista Studio’s programmer, Masa Takatsuka, has moved to the School of Information Technologies at the University of Sydney, Australia, and begun work on a new generation of Studio, called JBeanStudio. Check his website, http://www.cs.usyd.edu.au/~masa/research/JBeanStudio/html/news.html, and the Socnet beans website, http://orion.oac.uci.edu/~jastern/socnetbeans/ for more details.
4http://www.webgain.com/products/visual_cafe/
5http://www.borland.com/jbuilder/
6http://www7b.software.ibm.com/wsdd/zones/vajava/
Setup of a computer system for Studio and the socnet beans from scratch involves four basic steps (with some sub-steps):

1. Install Java Compiler (Java 2 Software Development Kit, or J2 SDK)
2. Install the Socnet and JDSL beans
3. Install Java Web Start (JWS)
4. Install/configure GeoVista Studio with socnet beans palette

Once the first three steps are taken, then when someone writes new beans, and you want to use them, then you will only have to go through the fourth step to add new beans to Studio.

Optionally, you may also want to perform one or more of the steps below:

5. Install other beans (such as Chem Symphony)
6. Install other external packages (such as Mage)

in order to view output files you create. I’ll show you how to do both of these, since we’ll need them for the tutorial section.

All six of these steps will be covered in this Installation section. The steps do assume you have Administrative or SuperUser/root privileges on your system.

### 2.1 Install the Java 2 SDK

The first step is to install the Java 2 software developer kit (SDK). Even though you may be using GeoVista Studio only from the standpoint of a user plugging beans together, GeoVista Studio still needs the Java 2 SDK to compile the connectors which work in the background to connect the beans together.
To get the J2SDK, point your web browser to

http://java.sun.com

click on Downloads, then Java 2 Platform, Standard Edition (J2SE)TM, then J2SE 1.4.1. Now there will be 2 columns – one for the Java Runtime Environment (JRE) only, and one for the Software Development Kit (SDK), which comes with the JRE included. Click on the SDK package for your architecture, agree to the terms, and download and install the package, according to the instructions.

On the same page with the JRE and SDK downloads, several lines down, are links to Installation instructions and README. Read both of these documents, and follow these instructions to install the SDK and the JRE that comes with it.

2.2 Install the socnet beans

You will need to install my beans as well as their support classes. Also, you’ll need a palette for GeoVista Studio, which tells Studio how to run these beans (how to set their defaults) without having to set all these defaults manually.

These files correspond to the three jar files and one palette file which need to be installed. In this section, you’ll download all four, but install only the .jar files. Later, when setting up Studio, you’ll use the fourth, the palette file.

2.2.1 Download the necessary files

The three jar files and the palette file may be retrieved from the socnet beans website at http://orion.oac.uci.edu/~jastern/socnetbeans/ These are the files you will want to download:

- jdsl.jar - the JDSL support classes
2.2.2 Tell your JVM where the jars are

There are two ways of telling your Java Virtual Machine (JVM) where the socnet jar files are: You can either drop the jar files into your JVM’s extensions folder, or you can put the jars somewhere else, and tell the JVM where this location is by adding it to an environmental variable called the CLASSPATH. Instructions are provided here only for the first method. While it is a less-well-known method, it is much easier. If you already know how to do these, skip to the next Section.

The extensions directory is under the JRE subdirectory, in the lib/ext folder. At the time of this writing, I can find this directory on my Linux system at

/usr/java/j2sdk1.4.1/jre/lib/ext

and on my Windows system, I can find the extensions folder at

C:\Program Files\Java\j2re1.4.1\lib\ext

Copy the three jar files into this location. If you need more help with this, or with using the CLASSPATH method, check out the New-To-Java-Programming Center at

http://developer.java.sun.com/developer/onlineTraining/new2java/

---

7 On Windows, Sun apparently installs two JRE’s: a second one is also installed at C:\j2sdk1.4.1\jre, and though not prominently-documented, it is, according to Sun, there for the java compiler to use for internal purposes. If you have problems at all, you might also try putting copies of the .jar files into C:\j2sdk1.4.1\jre\lib\ext as well.
At the time of this writing, directions can be found by clicking on Step 2: Getting Started, then Getting Started, by Dana Nourie. Look in the first few paragraphs for directions relating to “Next set up the software, and then configure your system.” Choose the item relating to your particular platform, Windows, Unix/Solaris, Mac, etc.

2.3 Install Java Web Start

GeoVista Studio runs from within the Java Web Start framework. Java Web Start – which you installed as the last step in the installation section – is a program that allows you to launch full stand-alone Java applications.

Each time you run it, your Java Web Start (JWS) will check to see if it has downloaded the latest version of the program you wish to run. It will do this by comparing the copy you automatically downloaded to your machine the last time you ran it to what is available at the program’s web site. If what is on the web site is not any newer, then JWS runs the program from a locally-cached copy kept on your hard drive from the last time you ran the program. If on the other hand, the program has been updated at its site on the WWW since the last time you ran it, JWS downloads the latest version automatically just before launching the program. Also, if your machine is temporarily disconnected from the WWW, then JWS just runs the local copy of the program.

The first time you run a JWS-based application, however, you need to get it

---

Java applications are different from java applets. Applets run only in browsers and have memory and file-access limitations. Applications are full-blooded programs which can access your hard drive (to save data) and can use more of your computer’s memory. JWS applications have the additional benefit of installing and running and updating themselves directly from their locations on the World Wide Web. Thus, Java Web Start achieves the benefits of both applets (by being runnable and auto-updateable from the web) and applications (by being more powerful).
into your JWS system’s list of installed (i.e., previously-run) programs. The way
to do this is to run the program from the web, as will be explained in Section 2.4.
After that, you can either run the program from the web again, or you will see
its icon appear in JWS’s window along with other applications you’ve run, and
you can click and run it from there.

2.3.1 Windows

If you have Windows, Java Web Start is installed with your SDK/JRE automatically, and a link is put on your desktop.

2.3.2 Linux

Setup of Java Web Start in Linux is a little more involved: In Linux, Sun dis-
tributes the JWS files as a .zip archive, but does not unpack and install them.
In my Linux box’s J2SDK, the zip is at

/usr/java/j2sdk1.4.1/jre/javaws-1_2-linux-i586-i.zip

This file came with your java distribution when you downloaded it.

First, make a directory where you’d like this to be, under your own home
directory. Then copy the zip file into it and unpack it:

% cd /home/jstern
% mkdir jws
% cd jws
% cp /usr/java/j2sdk1.4.1/jre/javaws-1_2-linux-i586-i.zip .
% unzip javaws-1_2-linux-i586-i.zip

Now run the install.sh installation shell script:

% ./install.sh
(It’s a bit of a crusty install script which will install JWS in a subdirectory right there, called javaws.) When it asks you the path of your installation, answer ’/usr/java/j2sdk1.4.1’. It will locate your SDK (and JRE in the jre subdirectory), and then it will finish its installation.

Now navigate to the javaws subdirectory created by the installation script:

```bash
% cd javaws
```

and check the ownership of the files it created:

```bash
% ls -l
```

```
total 872
-rw-rw-rw- 1 uucp 143 7910 Aug 26 21:55 cacerts
-r-xr-xr-x 1 uucp 143 816 Aug 26 21:55 javaws*
-rwrxr-xr-x 1 uucp 143 168384 Aug 26 21:55 javawsbin*
-rw-r--r-- 1 jstern jstern 213 Oct 2 07:44 javaws.cfg
-rw-r-xr-x 1 uucp 143 168384 Aug 26 21:55 javawsbin*
-rw-r--r-- 1 jstern jstern 213 Oct 2 07:44 javaws.cfg
-drwxr-sr-x 2 uucp 143 4096 Aug 26 21:55 resources/
% _
```

In my version of javaws, the writer of the shell installation script made the mistake of creating the files in the javaws directory be under user ID (UID) number 10 (which resolves to username uucp on my system) and group id number (GID) 143 (my system has no group with a number of 143). Therefore, you should change the ownership to yourself, such as I did (supply your own username/group here):

```bash
% chown -R jstern.jstern *
```

Now you should see:

```bash
% ls -l a
```

```
total 880
```

46
At this point, you should be set to run Java Web Start:

```
% ./javaws
```

Later, you can add a bash or sh alias to this program location by adding the following line:

```
alias jws='/home/jstern/jws/javaws/javaws'
```

to your shell’s startup script (.bashrc, .bash_profile). The same alias with a slightly different flavor:

```
alias jws ’/home/jstern/jws/javaws/javaws’
```

may be added to your .cshrc or .tcshrc startup script for csh and tcsh shells. This is so that you may type

```
% jws
```

from any directory, and it will always run the program.

Starting javaws should bring up a window which looks like Figure 2.1. Java Web Start starts with some sample applications already listed. You may try these just to get used to the system.
Now from the Java Web Start View menu, choose Downloaded Applications, as in Figure 2.2. This should give you a new set of applications available in Java Web Start. Mine looks like Figure 2.3, which includes not only GeoVista Studio, but two other applications. However, your set of downloaded apps will be empty.
the first time you run Java Web Start. We need to get GeoVista Studio into this list of downloaded applications, so that the next time, you can just click on it and then click Start, in order to start Studio.

The way to get GeoVista Studio into your Java Web Start is to run it the first time from the Web, as will be shown in the next section.

2.4 Install and Configure Geo Vista Studio

2.4.1 Installing

The next step is to install GeoVista Studio. Since installing happens automatically in JWS the first time you run a program, this is easy. To run GeoVista
Studio the first time, point your web browser to

http://www.geovistastudio.psu.edu

and click on the link to the right, Download Version 1.1 now. Go to the section labelled Step One and click on the button labelled, Launch.

When you first run GeoVista Studio, it should look like Figure 2.4.

![Figure 2.4: GeoVista Studio run for the first time](image)

2.4.2 Configuring

Studio needs to know only three things:

In Windows, the programs `javaws.exe`, `splash.exe`, `javaw.exe` all need to access the internet, so if you have a firewall going here, you may need to open it up for these programs. It is most common with Linux firewalls these days to let out-bound traffic have its way, so these problems should not normally present themselves. However if you are having difficulty in connecting with these programs, check your system’s `iptables` to determine whether they filter any outbound traffic.
• where to find your java compiler

• where to look for input and output data files (the working directory).

• what beans you want it to be aware of

2.4.2.1 Location of Java Compiler

While it is running, Studio runs the Java compiler. Therefore, Studio will need to know the location of the java compiler on your system.

To tell Studio where your java compiler is located, start up Studio, then click on Preference | User Preference, and in the User Preference dialog box that comes up, type in the location of your java compiler into the Java Compiler field, and hit OK. For instance, if you accepted the defaults in your Java2SDK installation,

![User Preference Dialog](image)

Figure 2.5: Telling Studio where your Java Compiler is

and you are in Windows, you would type in something like
C:\j2sdk1.4.1\bin\javac.exe

whereas in Linux\textsuperscript{10}, you would type something like

/usr/java/j2sdk1.4.1/bin/javac

You may see an example of this in Figure 2.5.

2.4.2.2 Location of Working Directory

In this preferences dialog box, you can also set the default Working Dir, or working directory. This is just the place where your Studio bean applications will look for input data files and save output files. Since we will want Studio to know where to look for any data files, let’s go ahead and set that now. Make a directory/folder anywhere on your system where you would like your working directory to be, and then type in the full pathname of that working directory here. We will use this working directory later in the tutorial.

Now your Studio is ready to install modules.

2.4.3 Import the Socnet Palette into Studio

The last step in setting up Studio is to make it aware of the socnet beans. There are two ways to do this.

One way is manually: create your own palette and bean folder, then load beans in manually, given a either a location or package path. Then, set up each bean’s default inputs, outputs and callbacks.

The other way is easier: If the person who created the beans also supplies you with a Studio palette file, and you have installed the beans where the provider

\textsuperscript{10}Your java compiler’s location will most likely vary slightly from the examples given here.
instructed, then you can import their palette file into the system, and you’re done. Studio uses the instructions in the supplied palette file to locate, load, and configure your beans automatically.

You have been provided with a palette file, socnet.xml, which you downloaded in Section 2.2.1. To import it into Studio, move the socnet.xml file to Studio’s working directory, then click on Palette, then Load Palette, then choose socnet.xml. You should now have a socnet palette, with a jastern bean folder, complete with the ten socnet_jastern beans, as seen in Figure 2.6.

![Figure 2.6: Socnet Palette and Beans in GeoVista Studio](image)

2.5 Installing Other Beans (e.g. ChemSymphony)

Though optional, it is highly likely you will want to install other bean-sets into Studio, besides my own. In fact, this is the whole point of using java beans collaboratively.

One such example set is the ChemSymphony beans [NetGenics Corp, 2002], created by NetGenics Corporation. ChemSymphony is a set of Java beans for analysis and visualization of chemical data structures.

ChemSymphony comes into play this way: One of the data formats used for chemical communication of datasets is PDB, short for “Protein DataBase Atomic Coordinate Entry Format” [Protein Data Bank, 1996]. The ChemSymphony beans will read PDB format. So since one of my socnet beans will translate a
Graph structure to PDB format, this means that one can use the ChemSymphony molecular-viewing beans to view your data as a chemical structure, from within GeoVista Studio itself (or any applet or application you create using it.). Thus, you don’t have to use an external viewer such as Mage to view your data in 3-D.

To download and install ChemSymphony, contact the folks at http://www.chemsymphony.com and download their demo. Install their jar much as you installed my socnet_jastern.jar beans, except that they do not have a palette available for you to download with the beans. Therefore you will have to create your own palette.

To install and configure Chem Symphony:

2.5.1 Install the ChemSymphony jar file.

Instead of copying the jar file into the lib/ext directory of your JRE, copy it into your Studio working directory.

2.5.2 Make a new Palette.

Click on Palette | New Palette and type in chemsymph.

ChemSymphony was a product of NetGenics Corporation, which — since the creation of this project’s software — has merged with Lion BioScience AG (Lion bioscience AG [2003]). ChemSymphony now appears to have been integrated into Lion’s iDEA software product (Lion bioscience AG [2002]). At the time of the first writing of this chapter they had an improved and much extended version 2 of the software, which was not shareware. At that time, version 1.3 was still made available to researchers for free under an academic end-user license agreement, and it is version 1.3 I will refer to here. Please contact Lion bioscience AG for further information. If you are not able to obtain a copy of ChemSymphony from them, it is not crucial to your use of the Socnet beans. Their use here is simply to show that alternative renderers may be used — and particularly, a Java-only renderer — at the end of a data flow pipeline to view a molecule.
2.5.3 Make a new Bean Folder.

Click on Palette | New Bean Folder and type in main.

2.5.4 Import the beans.

Click on Palette | New Bean, then click on From Jar File, then click on the Browse button. Select the chemsymphony jar file, as seen in Figure 2.7.

![Select Jar File](image)

Figure 2.7: Selecting the ChemSymphony Jar File

2.5.5 Select all beans.

Select all the beans found in the jar file, as seen in Figure 2.8. There are a lot of them, so remember to work your way down the entire list before clicking OK.
Figure 2.8: Adding all the ChemSymphony beans to GeoVista Studio

Your beans should now be included into Studio, as seen in Figure 2.9.

Figure 2.9: The ChemSymphony beans added to the palette
2.5.6 Manually configure the ChemSymphony beans

The ChemSymphony beans need to be configured for their inputs (setters, or red connectors) and outputs (callbacks, or blue connectors).

For the benefit of the tutorial in the next chapter, we will only configure the RenderBasic and RenderControl beans here. If you want to use any of the other beans, they will need to be configured, in a similar way.

2.5.6.1 Configure RenderBasic

Find the RenderBasic bean in the chemsymph palette, main bean folder (Figure 2.10), and click on it. Now click on the Configure Bean button (Figure 2.11).

![Figure 2.10: Using the mouse to find the RenderBasic ChemSymphony bean](image)

Click on the Input tab, and then, below that, click on RenderBasic. This should bring down a list of choices, and from these, choose GenericRenderer. From the list

---

12 using the mouse. It can be difficult to distinguish beans on the choiceChemSymphony palette, since many of these beans were not given unique bean icons by their creators. So you have to use your mouse, and pass over each one, looking at the hint text (as seen in Figure 2.10).
of possible inputs, choose these three: `propertyChange(PropertyChangeEvent)`, `setChemicalContents(String)`, and `setChemicalDataFormat(String)` (Figure 2.12). Make sure `render(RenderEvent)` is unchecked. Click on the Done button. Now whenever you drop a RenderBasic bean onto your design window, these three inputs should show up as red nodules on the bean tile (Figure 2.13).

### 2.5.6.2 Configure RenderControl

The `RenderControl` bean needs to be able to notify other beans when something has changed. Thus, we need to add an output (blue) node. Furthermore, when these other beans have been notified, they need to be able to call back and get the new information (a callback). So `RenderControl` has to be set up for both an output and a callback.

Find the `RenderControl` bean on the chemsymph palette (Figure 2.14) and click on the Configure Bean button as before.

Click the Output tabsheet and check `propertyChange.PropertyChange`
Figure 2.12: **RenderBasic** Properties Dialog: setting the three inputs (Figure 2.15).

Click on the **CallBack** tabsheet and check (PropertyChangeEvent)getIntegratedPropertyChangeEvent() (Figure 2.16).

That’s it for **RenderControl**. Close up the Properties Dialog by clicking on...
Figure 2.13: The RenderBasic Bean with 2 input nodules (on left-hand side)

Figure 2.14: Using the mouse to find the RenderControl ChemSymphony bean

Done.

Permanent configuration of bean inputs and outputs will be covered more completely in Section 4.2.3.6.

Section 4.6 will cover how the RenderBasic and RenderControl ChemSymphony beans may be used to visualize your social network data.

2.6 Installing External Software (e.g. Mage)

Lastly, Studio works well with other external software, as a “data translator” of sorts. Particularly, some programs work well as viewers of your datasets, but require their own formats.
Figure 2.15: Setting the **RenderControl** Output notification in the Properties Dialog

One such program is Mage\textsuperscript{13} by David and Jane Richardson. This program is an excellent chemical molecular viewer, but requires its own specialized data format (unfortunately, not PDB). An screenshot of Mage may be seen in Figure 2.17

One of the beans included in the `socnet_jastern.jar` jar package will translate a generic graph into Mage format (called a “kinemage”). Piping the output of this bean to another bean which saves output to a file, you can then use Mage to read in the file. Therefore, you can use Studio together with Mage to create

\textsuperscript{13}http://kinemage.biochem.duke.edu/
Figure 2.16: Setting the `RenderControl` Callback in the Properties Dialog

and view images rather quickly and easily.

Mage has the additional benefit of being somewhat cross-platform, as well, with viewers available for Windows/Intel, Linux/Intel, Macintosh (System 9) and SGI. There is also an Java applet viewer for publishing of your images on a website, if you like, though it does not have the full functionality of the standalone programs.
Figure 2.17: Mage
CHAPTER 3

Pretutorial: Background to Using the Socnet Beans

3.1 Introduction

The only purely-visual environment for connecting beans which emphasizes the flow of data between them is currently GeoVista Studio. The existence of Studio puts tremendous power into the hands of researchers without previous programming experience. Yet, in the end, using Studio is programming. It is just visual programming. When you connect two beans together visually with a line, you are telling one bean to give another bean some information when it becomes available.

This structure of connected beans in Studio starts with a design. You may save this design, which is simply a description (in an XML-style text file) of what beans to connect with what. In a design, no actual beans are saved, and no program is created. And yet, when you load a design, you are loading up beans and running them. Thus, a design may be thought of as a script, or a recipe. It is a text file listing which beans to use, along with their connections to each other, and their settings. When you load in a design you’ve saved earlier, you’re just reading the text file and following its directions to load in the beans you were using before, and connect them back up the same way.
Once the design is loaded in, you can see the beans’ connections (what might be termed the visual source code) in one of Studio’s windows, called the DesignBox. In another of Studio’s windows, called the GUIBox, you can see what the final Java program (applet or application) will look like. And that program is actually running as you view and design it.

Though it can sometimes be confusing, this ability to run at the same time you are designing is a major feature of Java and JavaBeans. And yet, it hadn’t been fully taken advantage of until GeoVista Studio. Studio’s creator decided to put a design window and a run window on your screen at the same time.

For researchers, this is optimal. You can drop beans onto your design window, connect them together, and get instantaneous results. You don’t need to compile the program and then run it. It already is running.

However, you can go further, should you like, and create a Java applet (for use on the World Wide Web), or a stand-alone Java application, which is meant to run outside a browser, on your computer, as a normal application.

Thus, with Studio, you can distribute your design to other Studio users, or you can distribute your applet on a web page of your own creation, or you can send someone an application which they can run with their own Java Virtual machine. You may even want to make your application run from Java Web Start, as Studio itself does.

As mentioned earlier, though Studio can be used to create almost any kind of design for a program, it is especially helpful for researchers who need to move their data through a special kind of design — a series of transformations I call a data flow pipeline design.

For this dissertation, I created enough beans to form 2 basic forms of data-flow pipelines based on Correspondence Analysis — one for analysis of 2-mode data,
and another for 1-mode data. The Studio designs for these pipelines demonstrate the efficacy of using a module-based approach to social network data analysis.

Chapters 4 and 5 are tutorials on how to create the Studio designs for these pipelines. Each pipeline starts by reading in a simple matrix of data that you supply, and it ends by creating a 3-D visualization of that data.

Chapter 4 runs the user through the creation of such a pipeline for 2-mode data. Chapter 5 demonstrates altering the pipeline created in Chapter 4 for 1-mode data. The tutorials assume you have installed all the software referred to in Chapter 2.

However, before starting these tutorials, it will be helpful if the reader understands some preliminary background concepts. These concepts include the object-oriented paradigm and its terminology; the basic Java data types and the syntax of Java method (procedure) calls; how these appear in Studio; Java beans and their basic inputs and outputs (methods); the Socnet bean as a specialized bean with a data processing cycle. These concepts will be covered in the current chapter.

3.2 Understanding Object Oriented Programming Terminology

Since beans are objects, it makes sense to first give a basic background for objects. If you understand objects already, skip to Section 3.3.

Objects were an invention to make the programming of complex applications easier. The idea was to break up an application into objects. Each object was a separate entity with its characteristics and abilities packaged together in the same place, instead of being spread out in different places (and not associated
together as an object) as before.

The characteristics of an object are called fields, and the abilities of that object, its methods.\footnote{\textsuperscript{1}}

Since you can have more than one of the same type of object, we call the type of object a class, and an actual instance of that class just an object, or sometimes an instance or instantiation.

To put this all together in an example of Java code, suppose we have a fictional Airplane class. Airplanes might have several variables, such as fuel_remaining and altitude.

The Airplane class might also have some methods, such as take_off, land, climb, descend, turn_left, and so on.

Thus, suppose a Java programmer had already defined the class, Airplane elsewhere, together with all its variables and methods. To create (instantiate) an object of class Airplane and give the object instance its own name (myplane), the programmer would type a line something like the following:

\begin{verbatim}
Airplane myplane;
\end{verbatim}

Then to tell the plane to take off if it has more than 3/4 of a tank of fuel, the programmer might write something like this:

\begin{verbatim}
if myplane.fuel_remaining > 0.75 then
  myplane.takeOff;
\end{verbatim}

and to fill up the plane the programmer might simply say:

\footnote{\textsuperscript{1}To a Pascal programmer, a field would be a fancy name for a variable, and a method a fancy name for a procedure. But there is an important difference for this nomenclature: in some object-oriented languages, variables and procedures can still exist along with fields and methods. The difference is that variables and procedures are not associated with — that is, put inside of — any object. Java does not allow free-form variables and procedures. Everything must be an object. Even the application itself is referred to as an object.}
myplane.fuel_remaining = 1.0;

Oh, if life were so easy.

Notice that in Java, the ‘.’ is used to refer to the fields and methods of an object, as if they were parts of that object – which they are. Both fields and methods are referred to the same way – with the ‘.’. Thus we can distinguish between telling myplane to take off,

    myplane.takeOff;

and telling another plane to take off

    mysecondplane.takeOff;

On the other hand, a reference to

    Airplane.takeOff;

might seem to be unclear. It is a reference to a class of object taking off. How can that be? This has special meaning in Java which is not necessary to know here. However, if you see it in Studio, it means that some instance of an Airplane is being told to take off. Studio has given the object instance its own name, of course, but Studio is not telling you the instantiation’s name because it is not important. Usually, you will know which object instance Studio is referring to because the reference came up in the context of the bean icon you just clicked on.

Classes of objects can descend from other classes. Thus, we could have a PropPlane class which descends from an Airplane class, inheriting its methods and fields (see Figure 3.1). Descended objects have all the abilities of their ancestral, or parental objects. However, descended objects can also have additional
methods that their ancestors don’t have, giving them extra functionality. For instance, suppose we had a PropPlane class which descends from (and therefore inherits all the functionality of) the Airplane class. So, suppose we had declared that PropPlanes have the additional ability of testing the functioning of their propellers, so we added the following method:

    boolean PropPlane.PropellersFunctioningOkay();

The 'boolean' at the beginning means that this method is going to return back an answer: either true or false value, depending on whether the propellers are indeed functioning.

Now, suppose we instantiated an object of this PropPlane class:

    PropPlane myPropPlane;

we could test these propellers, in our object:\footnote{\[
// set up our boolean variable
boolean propsOkay;
// now ask the plane if the props are doing okay,
// and save the answer into our boolean variable
propsOkay = myPropPlane.PropellersFunctioningOkay();
\]}

\footnote{The lines starting with ‘//’ are just comment lines, for clarity. They wouldn’t be interpreted by the Java compiler as real code.}
but since a PropPlane is an Airplane, we could still tell the plane to take off,

    myPropPlane.takeOff();

even though we had not specifically written a takeOff() method for PropPlanes.

Of course, other specialized classes could be descended from Airplane which inherit their abilities, but do other things, such as startJetEngines, etc.

### 3.3 Data Types and Method Declarations

As beans are objects, Studio refers to the methods of an bean in much the same way that a Java programmer would declare them in the code. Thus, while you don’t need to type in a declaration yourself, you should understand what one looks like. This will help you to understand how to hook the beans together in Studio.

#### 3.3.1 Data Types

Following are the data types you will most frequently encounter in Studio.

There are two groups of data-type. The first group is a list of *primitives*. Primitives are simpler than objects/classes. They are just built into the language itself. They may also be used as the fields of an object.

The second group of data types in Java are object classes. Classes may be further subdivided into those classes which come packaged with the Java compiler, and those which are written by third parties (such as the supporting classes needed for the Socnet beans, as well as the Socnet beans themselves).

---

3 compiled from [Horton, 1999, pp. 36, 40, 58]
Data types appearing most commonly in the Socnet beans are marked with a ‘*’.

3.3.1.1 Primitives

- **boolean** - Takes only 2 values: ‘true’ or ‘false’
- **byte** - An integer from -128 to +127, occupying 1 byte (8 bits) of memory.
- **short** - An integer from -32768 to +32767, occupying 2 bytes (16 bits) of memory.
- **int** - An integer from -2147483648 to +2147483647. Occupies 4 bytes (32 bits) of memory.
- **long** - An integer from -9,223,372,036,854,775,808 to +9,223,372,036,854,-775,808. Occupies 8 bytes (64 bits) of memory.
- **float** - A real number between \(-3.4 \times 10^{38}\) to \(+3.4 \times 10^{38}\), which takes up 4 bytes in memory, and having an accuracy of 7 digits
- **double** - A real number between \(-1.7 \times 10^{308}\) to \(+1.7 \times 10^{308}\), which takes up 8 bytes in memory, and has an accuracy of about 7 digits
- **char** - A single character, such as the letter ‘X’. Occupies 16 bits (2 bytes). Storage is in Unicode.

3.3.1.2 Java built-in Classes

- **Object** - Yes, there is a class of object called **Object**, just to keep us all confused. But actually, it is not too bad conceptually. It is just the parent of all classes of object. Thus, all objects descend from the class called
Object. Any class (such as an Airplane above, or String or Vector, as in Figure 3.2) is also a member of the Object class.

- *String* - An object which stores a string of characters, such as the string, 'hello there'.
- *Vector* - An array of objects.

### 3.3.1.3 Supporting Classes for the Socnet beans

Following are the classes of object that the Socnet beans most frequently expect to see as their inputs and outputs. These classes were tailored specifically to the types of data social network researchers would most commonly take advantage of.

- *Tensor* - An object storing a 3-dimensional matrix of doubles.

- *DL* - A Java object able to read in and parse a Data Language file on disk, and then hold its contents as a Tensor. Originally created for Analytic Language ("AL") [Borgatti (1985)], Data Language files are a common way of importing and exporting matrix data in UCINET [Borgatti et al. (2003)].
Figure 3.3: DL descends from (is a) Tensor

The DL class descends from the Tensor class. Thus, DL objects are also Tensors, having all their functionality. DL objects just have additional methods for reading in and parsing Data Language files, as well.

Therefore you may use a DL wherever you would use a Tensor. That is, you may connect the output of a ReadDL bean (which is an object of class DL) to the input of any other bean expecting a Tensor.

- *GraphArray* - An array of Graphs. A lowest common denominator Graph storage object, for importing in from and exporting out to other Graph types (such as Mage or PDB, etc.)

- *StringList* - An array of Strings.

### 3.3.2 Method Declarations

The above data types will appear in the method declarations that you see in Studio, when connecting beans. return-values and parameters (see following discussion) will be passed back and forth between the beans’ methods, and these return-values and parameters will be in the form of the above data types.

When you connect beans, you tell one bean which method to call in the other bean. The beans’ methods have been pre-declared by the programmers to expect certain data types coming and going (as parameters and return-values). Thus, in connecting beans, it makes sense to make sure that the methods match up in
terms of the data types they are sending out or expecting to come in. Therefore, it will help you to know not only the data types, but the Java notation for how these methods are declared, so you will know what each bean expects.

Beans may have both input and output methods. Both types of methods can take inputs (parameters) and give outputs return-values), though generally, output methods less commonly use parameters (input information) and input methods less commonly use return-value (output information).

Studio refers to both input and output methods in much the same way that a Java programmer would expect to see them in a code printout.

When a programmer declares a method for an object, such as

```java
void Airplane.takeOff()
```

or

```java
void Airplane.setFuelLevel(double)
```

or

```java
double Airplane.getFuelLevel()
```

There are four parts to understand:

```
return-value Class-name.method-name(parameters)
```

The return-value is information the method returns back to you. If you have a return-value of void, that means the method will do its work without returning any result data. Sometimes in Studio, reference to void is just omitted.

The Class-name is just the name of the class and the method-name is the name of the method. Sometimes, Studio omits the Class-name when the class
(or bean) being referred to is already obvious from the context (the bean you clicked on).

**Parameters** are information that you send to the method, so that it can do its work. There can be 0, 1, or more of these, though in Studio, you usually only see 0 or 1. If you have no parameters (`()`), that means the method can do its work without needing any input information.

A typical example of how GeoVista Studio uses the Java programming format of declarations may be seen in Figure 3.4. It is not important to understand fully

![Figure 3.4: Studio uses Java-style method declarations to communicate what data will be sent or received.](image)

Figure 3.4 at this point. However, notice the use of Java method syntax. At the top, Studio says the target bean is `ObjectToString`. The chosen input method
is `setInputObject` which expects one thing to be sent to it: some kind of `Object`.

Down in the lower right you can see a "getter" methods section, where there is only one choice, `(DL) getOutputDL()`. The full declaration of this in proper syntax would be:

```java
DL ReadDL.getOutputDL();
```

but Studio diverges a little from this syntax and just says:

```java
(DL) getOutputDL()
```

Studio omits the class name (ReadDL) because it knows that after a little practice you will know that when you connected `ObjectToString` to `ReadDL`, then when the Adapter Wizard window popped up, in the "getter" methods section, you are automatically being shown the data that `ObjectToString` can "get back" from the `ReadDL` bean.

Studio also puts parentheses around the return-value (DL), as a matter of style, apparently to remind you that this is data coming back – not a part of the `getOutputDL()` method.

But despite the slight divergences, you can see that Studio is telling you that when you connect `ReadDL` to `ObjectToString`, `ObjectToString` may call back to `ReadDL` for the `DL` object that it outputs.

Do not worry if you do not understand everything completely at this point. The process of connecting beans will be covered in more depth later in the chapter, and step-by-step directions will be given in tutorial chapters 5 and 4. It is only important that you get a first glimpse at Java and Studio notation for datatypes and methods so that you may connect beans together. As you get familiar with the syntax, it will become easier to rapidly make the connections between the beans.
3.4 Understanding Beans

To connect beans together, you need to know how they work. Beans are Java objects whose methods take a certain restricted form, so that they may be accessed by other pieces of code (beans or other normal code) which know nothing about them from the beginning. In the programming world, this is an example of what is known as *loose-coupling*, and it is crucial to the endeavor of cooperative computing. A programmer can create a bean and know that it will interoperate with other beans, without the programmer’s having to know how those other beans work.

Think of the methods as the coupling device on the cars of a train. The train works together because the cars, engine, and caboose are all designed so that they have the same coupling device on either side. The makers of the cars didn’t have to know anything about the other cars. They just had to make an agreed-upon connector.

Having an agreed-upon way of calling the methods works the same way. It means that environments like Studio can connect everyone’s beans together, but the writer of each bean doesn’t have to know anything about the internal workings of other peoples’ beans.

3.4.1 Setters and Getters for Properties

The foundation for a unified coupling (a way of calling methods) for beans is that beans have *properties*, and that properties may be accessed using *accessor* methods. These accessor methods are called *setters* and *getters*. These are simply methods with *set* and *get* prefixing the name, e.g., `setTemperature()` or `getTemperature()`, to set or get the current value of the `TEMPERATURE` property. This
simple convention allows Studio and other bean-building environments to inspect the bean and determine its methods at run-time so that it can be connected to other beans.

Thus, to rewrite our **Airplane**’s methods to be a simple JavaBean, a Java programmer might start with

```java
void Airplane.takeOff()
```

and rewrite it as a setter method:

```java
void Airplane.setFlying(boolean)
```

As users of this bean, to tell **myplane** to take off, we would say:

```java
myplane.setFlying(true)
```

and to land it, we would say:

```java
myplane.setFlying(false)
```

To fill the plane’s fuel, the programmer would have created a setter with one argument, a double-precision real-value:

```java
void Airplane.setFuelLevel(double)
```

so that instead of the old way,

```java
myplane.fuel_remaining = 1.0
```

with the JavaBean way we would say:

```java
myplane.setFuelLevel(1.0)
```
Setters often go with getters. Thus, to find out whether we are in the air, the programmer would have declared a getter:

    boolean Airplane.getFlying()

which we could use this way:

    // declare a boolean variable to store the result into
    boolean we_are_flying;
    // now get result and store it
    we_are_flying = myplane.getFlying();

which would return a true or false to us, depending on whether the plane was in the air or not.

Similarly, to find out how much fuel we have left, the programmer would have declared a getter in much the same way which returns a double instead:

    double Airplane.getFuelLevel();

and which we would use this way:

    // declare a double-precision real variable to store
    // the result into
    double fuel_remaining;
    // now get result and store it
    fuel_remaining = myplane.getFuelLevel();

In Javabeans, setters and getters together determine an implied property. Thus, with the getter and setter methods:

    double Airplane.getFuelLevel();

and

    void Airplane.setFuelLevel(double)
the implied property is FuelLevel.

However, not every property has to have both a setter and a getter. For instance, with JavaBeans, and with Socnet beans in particular, normally setters are used to set only input properties. Those input properties do not have getters, since there is no need, since the bean itself will retrieve the input value internally when it is ready to begin calculation.

Similarly, output properties – the results of a bean’s calculation – can be retrieved with a getter method, but normally will not have a corresponding setter method, since the bean itself sets the output value internally when it is done calculating.

If all this is too confusing, don’t worry about the particulars: the point is just that for the purpose of your working with Socnet beans, you may think of setters as being used to set inputs, and of getters as being used to get outputs.

3.4.2 The Socnet Bean Data Processing and Communication Cycle

There are several ways to connect JavaBeans together to transmit information. This flexibility makes beans useful in more environments, but at the same time, it can cause confusion for the researcher just wanting to plug beans together in a data-flow pipeline.

Fortunately, GeoVista Studio does accommodate all the various ways of connecting JavaBeans together, so that all JavaBeans – regardless of method of

\[\text{If the input property does have a getter, it will be only for a special purpose and will not normally be shown, i.e., will not appear as one of the getter methods when connecting up the bean’s blue output node. One such specialized purpose is “persistence” – the ability to save a bean along with the current state of all its fields and properties to a text file for later recreation – sort of like cryogenic freezing for later revival.}\]

\[\text{In like fashion, if the output property does have a setter, it will be only for a special purpose and will not normally be shown, i.e., will not appear as a red input node on the bean’s icon.}\]
communication—may be used. However, this generality still complicates things for the researcher, who simply wants to construct a dataflow pipeline for their research, and would rather have to learn only one simple model of bean communication.

Therefore, one of the main goals of this project was to unify the methods of the Socnet beans so that their use would be simplified in Studio, while retaining their usefulness in all other JavaBean environments.

Java beans—as mentioned in Section 3.1—can do anything. One bean might be a text window for you to type into. Another bean might be an alarm clock which rings at a certain time.

However, Socnet beans are a very specialized type of bean. All Socnet beans have the following in common: they filter, communicate, or translate data. Thus, Socnet beans are meant to be strung together in a data flow pipeline. This pipeline forms the basic structure for the visual Java program you are constructing.

Thus, because the Socnet beans form a pipeline, each Socnet bean uses the same simplified model for its functioning.

The core of this simplified model is the Socnet bean’s dataprocessing and communication cycle. In other words, how do we get data in and out of the bean? This cycle involves four steps:

1. Set the bean’s inputs with the bean’s setter methods

2. If the bean is not set to auto-calculate when all its inputs have been set, then manually tell the bean to start calculating by calling the bean’s do-
   Calculate() method

3. Wait for a message from the bean that it is done calculating. With Socnet
beans, this is always a PropertyChange event. With other beans it may be an ActionPerformed event, or other event.

4. Retrieve the result from one of the bean’s getters

To demonstrate this cycle, let’s suppose you had a fictional Socnet bean, myAdder, of the class Adder, for doing addition of two integers. The myAdder bean’s cycle of dataprocessing and communication would look like the following.

1. First, set the inputs. Set the first addend to 2 by calling its first input setter method:

   myAdder.setInputFirstAddend(2);

   Now set the second addend to 5 by calling its second input setter method:

   myAdder.setInputSecondAddend(5);

2. Then, you would tell myAdder to calculate:

   myAdder.doCalculate();

3. Now wait for myAdder to send out a message that it is done calculating:

   (a PropertyChange event is sent out)

4. Finally, call back for the result from one of the bean’s getters:

   // create an integer variable to save our total into
   int theTotal;

   // go out and get the total now
   theTotal = Adder.getOutputTotal();

82
At this point, the variable `theTotal` would contain the number 7. Notice also that the Socnet bean provides you with additional clues as to which are input and output methods, by having the words `Input` and `Output` in the naming of the setter and getter methods.

This is the Socnet bean data processing and communication cycle. It may be repeated as many times as necessary with each bean.

### 3.4.3 Connecting Socnet Beans

Connecting Socnet beans together to create a Studio design thus consists of getting the output from a source, or *upstream* bean (step 4, above) to become the input (step 1) of a target, or *downstream* bean, and then making the downstream bean calculate (step 2).

As far as the calculating (step 2, above), all beans except the initiating bean in a data flow pipeline design will calculate automatically when all their input variables have been set. The initiating bean will have to be manually told when to start. This is a rather straightforward process and will be covered in the tutorial.

As far as connecting outputs to inputs: With beans this can sometimes be rather complicated. However, in Studio this is handled rather easily, because Studio does a lot of work in the background for you. Though it is not necessary to go into great detail, you should know the basics of what Studio is managing for you, so when you try to connect up two beans, and it asks you for which *getter* method you want, you will know why.

When you create a connection between two beans, as in Figure 3.4, Studio creates an invisible object called an *Adapter* (shown in Figure 3.5), which sits

---

83

---

In Sections 4.6 and 5.11, where other beans are used which might have a slightly different method-naming style, the reader will be guided through their usage.
Studio connects beans with an invisible adapter between the two beans and serves as a data conduit. Each adapter goes through the following steps in the background, in order to communicate data from one bean to the next for you, which you can see as steps 1-3 in Figure 3.5.

1. Receive the the **PropertyChange** event message from the upstream bean that it is done calculating

2. Call back to the upstream bean’s output getter method to get the result, and store it

3. Now call the input method of the next bean, giving it the result data from the previous bean

In Studio, when you first connect an upstream bean’s output (‘done-calculating’) message (**PropertyChange** event) to a downstream bean’s input setter, Studio will need help setting up this adapter. So right then, Studio will pop up a window like the one you see in Figure 3.4 asking you what getter from the upstream bean you would like to use to get the result.
Thus, in the case of Figure 3.4 you can see that for the upstream ReadDL bean, the only getter that is available is \texttt{getOutputDL()}, which returns the output of the ReadDL bean: a DL object. Once this getter method is chosen, the Adapter Wizard in Figure 3.4 will set up the adapter in the background, so that the DL object it gets from \texttt{ReadDL.getOutputDL()} will be sent to the downstream bean, \texttt{ObjectToString}, as data (the Object parameter) for its input setter method, \texttt{ObjectToString.setInputObject(Object)}. Remember, since any object of class DL is also a member of class Object, \texttt{ObjectToString} will accept a DL for input to its setter method, \texttt{ObjectToString.setInputObject(Object)}.

Although it may seem complicated in the abstract, this really is just a matter of simple mouse-click for the user. After that, though there is an invisible adapter object connecting them, all you see is a line connecting the beans.

The actual process of connecting beans into a dataflow pipeline Studio design will be detailed in Chapter 4.
CHAPTER 4

Tutorial Part 1: Correspondence Analysis of Two-Mode data with the Socnet Beans

4.1 Introduction

This tutorial will lead you through creating a data-flow design in GeoVista Studio in order to visualize 2-mode data. The design will create at the end an image which can be viewed in the Mage molecular viewer. This image will show both the rows and columns as distinctly colored sets of spheres in 3-d space.

The dataset we will use comes from Weller and Romney (1990, pp. 55–62) and is distilled from data produced by Srole (1962). This is a breakdown of broad categories of mental health by socio-economic status (SES), from people in midtown Manhattan, in New York City. It may be seen in Table 4.1. In this table, the columns (SES) describe the socio-economic status of the parents of the

<table>
<thead>
<tr>
<th>Mental Health</th>
<th>SocioEconomic Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AB</td>
</tr>
<tr>
<td>well</td>
<td>121</td>
</tr>
<tr>
<td>mild or moderate</td>
<td>300</td>
</tr>
<tr>
<td>impaired</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 4.1: Srole Manhattan SES/Mental Health Data
individuals polled in midtown Manhattan. The rows describe the individuals’ mental condition. At this point in the text, Weller and Romney have combined mild and moderate into a single category. SES has also been combined, based upon previous analysis. The cells contain simple frequencies.

Although this is a very simple example, we are simply interested in creating a graphic which will shed some light on the question: Is there a relationship between one’s mental health, and one’s parents’ socio-economic status (SES)?

We are going to start with a simple DL-formatted text file of this data which

```
dl nr 3 nc 4 format fullmatrix diagonal present
title "Parental SES by Mental Health"
column labels:
  AB
  CD
  E
  F
row labels:
  well
  mildmoderate
  impaired
data:
  121 129 36 21
  300 388 151 125
  86 154 78 71
```

Figure 4.1: DL-formatted SES/Mental data

you can see in Figure 4.1. You can either create this file manually or download it from the socnet beans website.

When we are done, you will have a design which creates a description of an image in Mage (kinemage) format and saves it to a file, which can then be opened by Mage, as seen in Figure 4.55. So if you want to view the file you created as a
3-D image, make sure that in addition to the normal GeoVista Studio and socnet beans installation, you have Mage installed, as directed in Section 2.6.

4.2 Read In the Data

First we will start with creating an input file and putting some beans down in Studio.

4.2.1 Create the input file

First, create a srole folder/directory for working with the Srole data, and put it wherever you like.

Now manually create a text file named sesmental_dl.txt whose contents look like Figure 4.1. (You may also be able to just download this file from http://orion.oac.uci.edu/~jastern/socnetbeans/) Put this file into your srole directory.

4.2.2 Read in the input file

4.2.2.1 Drop a ReadDL bean onto the DesignBox

We need a bean to read our DL file in. The bean to do this is the ReadDL bean.

Start GeoVista Studio from JWS, if it is not already started. Choose the Socnet palette. In the jastern bean folder, click on the ReadDL bean, then click again in the Design Box. You should now see something like Figure 4.2. Notice also that in the GUIBox to the right, the ReadDL bean does not appear as a visual component, except to have a marker which somewhat resembles a button. Visual and non-visual beans will be discussed in the next section.
4.2.2.2 Add MtSimpleFileChooser

Now we have something to read in a DL file, but we need to tell it which file to
read in. There are several ways to do this. One of the easiest is to use Studio’s
SimpleFileChooser bean. This is back on the System palette, in the I/O bean
folder. Go back there, click on the SimpleFileChooser icon, and click it onto
the DesignBox window, in the same way you did with the ReadDL bean.

In the right hand side of Studio, in the GUIBox, you will notice that there is
now an entry box that is labelled File Name:, and it has a Select button. This is the MtSimpleFileChooser bean’s visual component. That is, this is the way
the MtSimpleFileChooser bean will appear to the user of the final application,
should you end up compiling the application. Notice that the MtSimpleFile-
Chooser bean is a visible bean, while the ReadDL socnet bean is an invisible
ReadDL works in the background, but will not have a visible appearance in the final form. If you think about it, this makes sense: the ReadDL bean simply reads in a file and translates it. There is nothing in what it does for the user to interact with. Visible beans have visual components while invisible beans do not have components.

Now re-size the MtSimpleFileChooser’s visual component so that it is a good size so that you can read what you type in in the window (Figure 4.3). Resizing a bean’s visual components in Studio’s GUIBox can be a challenge at first, because the interaction is somewhat opposite what you might expect\(^1\). In the GUIBox you drag on the corner of the component to resize it and on the side or top/bottom of it to move it. You cannot drag on the middle of a component. You have to be careful to click and drag on the edge. Experiment a little until you feel comfortable with resizing and moving components.

\(^1\)The reason for this has to do with the fact that beans are running at the same time you are designing with them, so design-related mouse movements had to be distinguished from run-time-related mouse movements.
4.2.2.3 Connect MtSimpleFileChooser to ReadDL

Now you need to hook the two beans together. The reason you do this is so that when you select a file, the SimpleFileChooser bean will tell the ReadDL bean the name of the file you chose, to load in.

You will notice that the ReadDL bean has two red ‘input’ connectors. These represent input methods (‘setters’). If you leave your mouse over these connectors, you will see that one is called doCalculate() (Figure 4.4), and the other is called setInputDLFileName(String) (Figure 4.5). You may notice that when you run Studio, for instance, doCalculate appears as the bottom red node and setInputDLFileName(String) appears at the top. While it may throw off users a little at the beginning, this behavior does not in any way affect the processing of the data. All that matters is that you can find which red node stands for which of the bean’s methods.

The blue right-hand side connector is the output connector (Figure 4.6). As

2Sometimes Studio rearranges the order of these red connectors, when it loads in a design. You may notice that when you run Studio, for instance, doCalculate appears as the bottom red node and setInputDLFileName(String) appears at the top. While it may throw off users a little at the beginning, this behavior does not in any way affect the processing of the data. All that matters is that you can find which red node stands for which of the bean’s methods.
you may recall from Sections 3.4.2 and 3.4.3, the Socnet beans tell you when they are done by sending an event, or message. The blue connector actually represents a place for you to connect to, to listen for this event. This event is a message to input connectors from other beans saying, “I’m done calculating. Come pick up your output (data) now.”

On the Socnet beans, this output event is a propertyChange.propertyChange event, but other events may be used with other beans. On components provided by other bean providers (including the beans built in to Studio) these can be many other things, but are most often an action.actionPerformed event. Such is the case with Studio’s own MtSimpleFileChooser bean, which uses the actionPerformed event (Figure 4.7) to signal that a file has been chosen. An actionPerformed event serves the same purpose as a PropertyChange event: it just notifies the next bean that data is ready. So do not worry too much about the nomenclature of these events. I will let you know if there is anything special to watch out for.

On the MtSimpleFileChooser bean’s visual component, you thus see the a little blue connector on the right side: it says it is an action.actionPerformed event, or message (Figure 4.7). MtSimpleFileChooser will send the action-.actionPerformed event every time you hit the Enter or Return key in the File
Figure 4.7: DesignBox: Finding the `actionPerformed()` output node on the MtSimpleFileChooser bean

Name field of the MtSimpleFileChooser’s visual component. That is, you can enter a file name and hit Enter, and the MtSimpleFileChooser bean will send out the `action.actionPerformed` signal that new data has been entered.

Thus we can use MtSimpleFileChooser to choose the input file: every time we make a new entry in the File Name field and hit Enter, the ReadDL will be updated with the new name of the input file for it to read.

Connect MtSimpleFileChooser’s `action.actionPerformed` output connector to the ReadDL bean’s `setInputDLFileName(String)` input (setter) connector (Figure 4.8). Put your mouse on the MtSimpleFileChooser bean’s blue `action.actionPerformed` connector, and then, while holding down the left mouse button, drag the mouse up to the ReadDL bean’s red `setInputDLFileName(String)` connector. When you do this you will see a red line. When you’re done, let go.

Now you will see an ‘Adaptor Wizard’ dialog box (window) pop up, as in Figure 4.9. This Adapter Wizard is asking you which getter you would like to use from MtSimpleFileChooser as an output. You can see that MtSimple-
FileChooser offers two getters – two ways of getting the data out from the File Name field: one getter, `getFile()` will give you a File object. The other, `getAbsolutePath()`, will give you a String object.

Since the ReadDL’s input setter, `setInputDLFileName(String)` expects an object of class String string as its input, we should use the `getAbsolutePath()` getter. Click on this, as in Figure 4.9, and then click on Finish.

Now, back in the Studio DesignBox, you should see the two beans connected by a pipe (Figure 4.10).

### 4.2.2.4 Connect JButton to ReadDL

What will start the beans executing? In order to start the data flow network, we have to manually tell the initial bean (in this case, ReadDL) to start calculating. The way to do this is to call ReadDL’s `doCalculate()` method. But how may we do this with visual components? The easiest way to do this is with a button the user may click on.

GeoVista Studio provides just such a component, JButton, which we will use. Go to Studio’s System palette, and the GUI bean folder. First on that
Figure 4.9: The Adapter Wizard for connecting outputs to inputs

palette should be the **JButton** bean.

Click on the **JButton** bean in the bean folder and then once again somewhere in the Studio DesignBox.

Now connect **JButton** to **ReadDL** as you did before: by dragging the mouse from **JButton**’s blue output connector to **ReadDL** free input *doCalculate()* input connector (Figure 4.11).
Now a very simple Adapter Wizard window will pop up, without any choices for getter arguments, as seen in Figure 4.12. This is because the ReadDL bean’s `doCalculate()` method requires no arguments (parameters). So just click Finish. The JButton is connected to the ReadDL (Figure 4.13). Now whenever the user clicks on the JButton visual component, it will cause ReadDL to execute its `doCalculate()` method, and start calculating.
4.2.2.5 Change the label on the JButton

Resize and move the JButton’s component in Studio’s GUIBox to your liking, as was done with the MtSimpleFileChooser component in Section 4.2.2.2.

Now observe that your JButton’s component does not have any text. Let us put the word “Calculate” on the button itself, so it is clear what this button
Figure 4.13: DesignBox: JButton connected to ReadDL

Right-click on the JButton in the DesignBox, and click on Property (Figure 4.14). This will open up a Properties dialog box for the JButton (Figure 4.15). Now scroll down to the Text property in the Properties tab. Click on it, and type in the word Calculate. Be sure to hit the Return or Enter key after you type in the word Calculate. Otherwise the change will not register. This is an idiosyncracy of Studio.

Click Done and resize your JButton’s component in the GUIBox. Your
Figure 4.15: JButton Properties dialog: Entering a label in the Text property

GUIBox should now look something like Figure 4.16.

Figure 4.16: GUIBox: JButton with new text label
4.2.3 Test that you’ve read in the DL file

To verify that you’ve read in the Tensor, you can print it back out as a String object. To do this, you will need two more beans: ObjectToString and JTextArea.

ObjectToString is a socnet bean which just converts any object to a String representation.[3]

JTextArea is a bean which comes with Studio. It provides a window for large amounts of text to be viewed.

So, using ObjectToString to convert the Tensor to a String, and then JTextArea to view the String, you will be able to verify the data you read in.

4.2.3.1 Add an ObjectToString bean

Go back to the Socnet palette and the jastern bean folder. Find the ObjectToString bean, and put one on the Studio’s DesignBox.

4.2.3.2 Add a JTextArea bean

Now go back to the System palette and the GUI bean folder and find the JTextArea bean. Put one of these on the Studio DesignBox, as well.

Over in your Studio GUIBox, make sure you resize the JTextArea’s visual component so that it is a reasonable size.

[3] Actually, ObjectToString does not do any conversions. Instead, it just calls the object’s toString() method, and passes on the String it receives back.

In the case of sending it a Tensor, ObjectToString just asks the Tensor to give it back a String version of itself. It does this by calling on (String)Tensor.toString() in the background. Then it just passes on the String it received from the Tensor.
4.2.3.3 Connect ReadDL to ObjectToString

Connect \((\text{Tensor})\text{ReadDL}.\text{getOutputTensor}()\) to \(\text{ObjectToString}.\text{setInputObject}(\text{Object})\).

That is, connect the ReadDL’s output connector, \(\text{propertyChange}.\text{propertyChange}\), to ObjectToString’s single input connector, \(\text{setInputObject}(\text{Object})\). In the Adapter Wizard, choose \((\text{Tensor})\text{getOutputTensor}()\).

Now your DesignBox should look something like Figure 4.17.

![DesignBox Diagram](image)

Figure 4.17: DesignBox: \text{ObjectToString} and \text{JTextArea} added

4.2.3.4 Change JTextArea’s interface

Now we have to detour for a bit and set up the \text{JTextArea} to run a little differently. Take a look at \text{JTextArea}’s single red input connector. If you rest your mouse pointer over this connector, you see that the input method name is \text{append(String)}.

There are two things to notice here. The first is that this method is not \(\text{(DL)}\text{ReadDL}.\text{getOutputDL}()\): we want the \text{Tensor}, not the DL object. Since a DL is a \text{Tensor}, this would work, but a DL prints itself as a \text{String} differently, making it more difficult to verify in the \text{JTextArea}. Otherwise their behavior in this design would be exactly the same.
technically a setter method, since it does not start with the word *set*. However, you can still connect to it as an input method. Some non-Socnet beans do this with their input and output/call-back methods. You can still connect to these and use them as setters and getters, but you will have to be more careful in their use.\footnote{Deviation from the set/get convention for bean methods does add to a bit of confusion. As mentioned in Chapter \[\ref{chp:3} \] the Socnet beans, in order to remain clear, do not deviate from the set/get terminology. Furthermore, wherever possible, Socnet bean methods include in their names the words 'Input' and 'Output' in order to remind the user of their purpose – e.g. ReadDL’s getter method, *getOutputTensor()*.

The second thing to notice concerns *append(String)* itself. As its name implies, whenever a *String* is sent to the *JTextArea*, the *String*’s contents will be *appended* to whatever is already in the *JTextArea*’s window.

But what if we want not to *add* to the window, but instead *replace* its contents entirely? Should not *JTextArea* allow us to do that?

It turns out that *JTextArea* does have a method for this, called *JTextArea.setText(String)*. However, Studio does not by default have *JTextArea* set up to *show* this method as a red input connector. Therefore, we have to set up *JTextArea* in Studio to add a second red input connector for *setText(String)*.

4.2.3.5 Changing the interface for this *JTextArea* instance only

In Studio, there are two ways to add or subtract input connectors: one is for the current instance only (the bean you have dragged out onto the DesignBox) and the other is for all future beans you drag out. I will show you both.

To add the input method for this instance only, right-click on the *JTextArea* that you put on the DesignBox, and choose *Properties*, just as you did with the *JButton* in Section \[\ref{sec:4.2.3} \]. You should see the *JTextArea* properties dialog.
window come up, as in Figure 4.18. Click on the Input tab, to get to the choices of

![JTextArea Properties Dialog Box](image)

**Figure 4.18: JTextArea Properties Dialog Box**

Input methods you want to show on the bean. (Figure 4.19). Notice that there is only one input method checked: `append(String)`. This is the red input connector you see on the outside of the current JTextArea bean on the DesignBox.

If you scan the list of other methods available, you will notice that the `setText(String)` is not among them. That is because `setText` is not actually a method belonging to JTextArea itself. It is a method of JTextArea’s immediate ancestor class, JTextComponent. But, as explained in Section 3.2 because JTextArea is a descendant of JTextComponent, it inherits all the
same methods (functionality) of its ancestor.

How do we access the ancestor method? Looking at the window, notice there is a bar that says, JTextArea. If you click on this bar, you will get a drop-down list of the bean and all its ancestor classes. This functionality allows you to see the input methods not only for this bean, but also for all its ancestor classes. We want to see the input methods available from JTextComponent, so choose that (as in Figure 4.20). Now if you scroll down to the bottom of that list, you should be able to select setText(String) as seen in Figure 4.21. Choose Done to close the Properties dialog window. Now you should see that your JTextArea bean has
two red inputs (Figure 4.22). If you rest your mouse over the lower of these two connectors (Figure 4.23), you should be able to verify that it corresponds to the `setText(String)` method you have just added. The input method, `setText(String)` is now added for the current `JTextArea` instance.

4.2.3.6 Changing the interface for all future JTextArea instances

To change Studio so that all future instances of `JTextArea` which you put on the DesignBox have the `setText(String)` input method (red connector) as well,
Figure 4.21: JTextArea Properties dialog: Adding the setText input method from JTextComponent

Figure 4.22: DesignBox: JTextArea with second input connector added

the process is only a little different.

First, go to the System palette and the GUI bean folder. Then, click on the
**Figure 4.23:** DesignBox: Verifying the `setText(String)` method input connector

**JTextArea** icon in the bean folder, as in Figure 4.24. Now click on **Configure**

![Figure 4.24: Selecting the JTextArea bean in the bean folder](image)

**Bean**, as in Figure 4.25. This will bring up the same properties dialog window

![Figure 4.25: Configuring all future instances of the JTextArea bean](image)

as you got before (Figure 4.18) when setting properties for only the current **JTextArea** instance. However, setting any of these properties now will affect all future instances of **JTextArea** beans you put on the DesignBox.
At this point, click on the Input tab and go through the same procedure you did in Section 4.2.3.5. This will add the `setText(String)` input connector for all future `JTextArea`’s that you put on the DesignBox.

### 4.2.3.7 Connect `ObjectToString` to `JTextArea`

Now we are ready to continue connecting up the `ObjectToString` with `JTextArea` so that `JTextArea` can print out the `Tensor` (as a `String`) in its window.

Connect `(String)objectToString.getOutputString()` to `JTextArea.setText(String)`.

That is, connect `ObjectToString`’s blue output connector, `propertyChange.propertyChange` to `JTextArea`’s red input connector, `setText(String)`. In getter methods in the Adaptor Wizard, choose the default and only choice, `(String) getOutputString()` and click the Finish button.

Now you should have a design which looks like Figure 4.26.

![DesignBox: First design finished](image)

**Figure 4.26: DesignBox: First design finished**

### 4.2.3.8 Select file

Now we have a ready design, but we need to tell it which input file to use.
Click on the MtSimpleFileChooser’s component’s Select button, navigate to the directory where you saved your ses\_mental\_dl.txt file, and choose it. Now you should have the file name showing in the component, as in Figure 4.27.

![File Name: /home/jstern/dissert/dev/testfiles/srrole/sas\_mental\_dl.txt](image)

Figure 4.27: GUIBox/MtSimpleFileChooser: Input file chosen

### 4.2.3.9 Run the design

Now just click on the JButton (labelled Calculate) in Studio’s GUIBox, and you should see the Tensor come up as a String in the JTextArea’s window (Figure 4.28). This is a Tensor version of the DL file you read in, and it verifies that we’ve got data flowing through the design.

### 4.2.3.10 Disconnect ReadDL and ObjectToString

Disconnect the ReadDL and ObjectToString beans when you are finished. To do this, very carefully put your mouse on the pipe connecting them (Figure 4.29), and right-click. (If you get a pop-up menu that has only 'View' and 'Show Module Name' on it, you’re not clicking right on the pipe itself. Reposition the mouse and try again.)

In the pop-up menu, left-click on Delete (Figure 4.30). The connection has been broken.
4.2.3.11 Move ObjectToString and JTextArea aside

Now move ObjectToString and JTextArea (which are still connected to each other) to the lower-right side of the Studio DesignBox. We will use these later.
4.2.3.12 Save your design

We’ve done a fair amount of work to this point. This is a good place to save our design. But where to save it? One way is to set up a separate directory for your designs, since the same design may be used on more than one dataset. Another way is to make a copy of the design and put it in each dataset’s directory. This way, you remember which design you used on which dataset. Either method has advantages. We will use the second: store the design to the same directory our Srole data is in.

First, click on Studio’s File menu, and choose Save Design, as in Figure 4.31. This will bring up a Save Design dialog window (Figure 4.32). In the File Name field, type in the name of your design, `2modeviewer`. Studio will append `.gvd`
Figure 4.31: Telling Studio you want to save your design

Figure 4.32: Studio’s Save Design dialog box
to the end of the file name automatically, so that it is actually saved as `2mode-viewer.gvd`.

### 4.3 Correspondence Analysis: Two-Mode

Let us continue on with the tutorial, creating a design which will end with an image file we can view with an external image viewer, to view our SES Mental data as a 3-D image.

To do this, we will need $x$, $y$, and $z$ coordinates for each of the row and column items so that we can plot them in 3-D space. So how can we generate an \{$x, y, z$\} coordinate for each row and for each column, and plot them on the same scale in 3-D space? One of the most popular ways is by performing a Correspondence Analysis (CA) on the original data. There are four steps to a Correspondence Analysis. The core step is to do a Singular Value Decomposition, or SVD, of the input data. CA adds another pre-SVD step of normalizing the data (to remove marginal effects), and 2 post-SVD steps (so that rows and columns may be viewed in the same space). The CA will provide a set of dimensions found in the matrix. It will also provide a set of “row-scores” — how each row scores on each dimension. It provides the same for the columns. We will take the scores (on both rows and columns) from the first three of these dimensions, and use these scores as $x$, $y$, $z$ coordinates in 3-space.\(^6\)

Once we’ve obtained our coordinates (in the form of another Tensor) we will convert them into a generic type of graph object called a GraphArray. Then we will convert the GraphArray out to a String which contains the description

---

\(^6\)CA and other similar methods are explained for social scientists in Weller and Romney (1990, pp. 55–62). A more technical explanation of CA only, with examples, is given in Greenacre (1984).
of the picture we want to see in Mage format\textsuperscript{7} so that we can later save this Mage-formatted String to a file and read it back in with the Mage program itself.

4.3.1 Normalize the data

4.3.1.1 Add SqRtNorm

We are ready to put in the beans necessary to perform a Correspondence Analysis of the Srole SES/Mental data and give us the 3-D coordinates we need.

The first step of CA is to perform a pre-SVD normalization on the data. To do so, go back to the Socnet palette / jastern bean folder. Find the SqRtNorm bean and put an instance onto the DesignBox.

4.3.1.2 Connect ReadDL to SqRtNorm

We need ReadDL to send SqRtNorm a Tensor.

Connect (Tensor)ReadDL.getOutputStream() to SqRtNorm.setInputTensor(Tensor). In other words, connect ReadDL’s propertyChange.propertyChange output connector to SqRtNorm’s setInputTensor(Tensor) input connector. Choose “getter” method (DL)getOutputTensor(). ReadDL will from now on give the SqRtNorm bean its output Tensor.

\textsuperscript{7}called by Mage’s creators a kinemage
4.3.1.3 Test your connection to SqRtNorm

Temporarily connect up SqRtNorm to ObjectToString, which we left on the DesignBox from our earlier work. See if you can do this. Your design should look like Figure 4.33.

![DesignBox: Connecting the SqRtNorm bean to ObjectToString](image)

Figure 4.33: DesignBox: Connecting the SqRtNorm bean to ObjectToString

Now click on the Calculate button again. This should result in your JTextArea window showing some new output, as in Figure 4.34. What are you seeing in the JTextArea window? What is this? What class of object does SqRtNorm output? Why did ObjectToString accept this as its input?

Also, do you notice anything about the size of the numbers? What about the dimensions of the matrix?

Disconnect SqRtNorm from ObjectToString, and move ObjectToString and JTextArea back to the side, to make room for more beans.

Remember to save your design.
4.3.2 Perform the Singular Value Decomposition

4.3.2.1 Add SVD bean

Now we need to do an SVD on the normalized data. Put an SVD bean (from the Socnet palette) onto the DesignBox to the right of SqRtNorm.
4.3.2.2 Connect SqRtNorm to SVD

Now that SqRtNorm has performed its normalization on the input Tensor, we need to send the new, normalized Tensor to SVD so SVD can do a singular value decomposition on the normalized Tensor.

Connect (Tensor)SqRtNorm.getOutputTensor() to SVD.setInputTensor(Tensor).

That is, connect up SqRtNorm’s output (the usual propertyChange.propertyChange output event) to SVD’s input, setInputTensor(Tensor). In the Adapter Wizard, choose SqRtNorm’s default ”getter”, (Tensor)getOutputTensor(). So we are getting SqRtNorm’s output Tensor object (which is the normalized data from the ReadDL), and we are giving it to the SVD routine.

4.3.2.3 Test SVD’s outputs

Temporarily connect up SVD to ObjectToString, as we did with SqRtNorm in Section 4.3.1.3.

Notice the several getter methods (outputs) that SVD offers (Figure 4.35). Each of these getters returns a Tensor, as did SqRtNorm. What the returned Tensor contains however, differs for each getter:

- getOutputD() returns an output Tensor containing the singular values of the input Tensor
- getOutputDStats() returns an output Tensor containing the same singular values of the input Tensor along with some cumulative statistics
- getOutputRanks returns an output Tensor containing the ranks (one rank for each matrix in the input Tensor).
**Figure 4.35: Adapter Wizard: SVD’s available outputs (getters)**

- `getOutputU()` returns an output `Tensor` containing the row scores of the input `Tensor`
- `getOutputV()` returns an output `Tensor` containing the column scores of the input `Tensor`

Try connecting each of these getters to `ObjectToString` and clicking on `Calculate`
each time. Then disconnect and try the next one. What do you see in the

```
title = "Row Scores of Parental SES by Mental Health, sqrt normalized"
nummatrs = 1
numrows = 3
numcols = 3
matrixlabels = "mat1"
rowlabels = "well" "mild/moderate" "impaired"
collabels = "dim0" "dim1" "dim2"
data =
0.43004622895235856 -0.7017055305676678 -0.5680401300388935
0.7620517643613622 -0.65523270598356456 0.6451561490239924
0.48408403133917793 0.7103229522973903 -0.5109833207081509
```

Figure 4.36: GUIBox/JTextArea. Square-root normalized row scores, as a Tensor printing out as a String

**JTextArea** component window each time? Look over how the dimensions and data differ. Figure 4.36 shows an example of the output for `SVD.GetOutputU()`, in the **JTextArea**.

Be sure to disconnect **SVD** from **ObjectToString** when you are finished. Save your design.

### 4.3.3 Scale the row and column scores

#### 4.3.3.1 Add OptScale bean

Now we need to optimally scale the row and column scores from the singular value decomposition. To do this, start by putting an **OptScale** bean onto the Studio DesignBox.
4.3.3.2 Connect ReadDL to OptScale

An optimal scaling rescales the row and column scores of the output of the SVD, according to (the dimensions) of the original data matrix. Thus, the OptScale bean needs three input Tensors: the original data Tensor from the ReadDL bean and the SVD row and column scores from the SVD bean.

Connect (Tensor)ReadDL.getOutputTensor() to OptScale.setInputO(Tensor).

Now OptScale can get the original data from the ReadDL bean.

4.3.3.3 Connect SVD to OptScale

Now OptScale needs to get the row and column scores (U and V matrices, in SVD parlance) from the SVD bean.

To send the row scores from the SVD, connect (Tensor)SVD.getOutputU() to OptScale.setInputU(Tensor).

To send the column scores from the SVD, connect (Tensor)SVD.getV() to OptScale.setInputV(Tensor).

Your design should now look like Figure 4.37. Again, do not worry if the

Figure 4.37: DesignBox: OptScale added
order of the red input connectors on \textbf{OptScale} is different in your design.

4.3.3.4 Test OptScale’s outputs

Temporarily connect up \textbf{OptScale} to \textbf{ObjectToString}, as we did in Sections \ref{sec:4.3.1.3} and \ref{sec:4.3.2.3}.

As with \textbf{SVD}, \textbf{OptScale} offers more than one output. Notice in the Adapter Wizard (Figure \ref{fig:4.38}) that you can choose from \texttt{getOutputX()} and \texttt{getOutputY()}. These are the optimally scaled row ($x$) and column ($y$) scores. Choose \texttt{getOutputX()} and click on the the \texttt{Calculate} button in Studio’s GUIBox. You should now get output that looks like Figure \ref{fig:4.39}.

If you like, disconnect \textbf{OptScale} from \textbf{ObjectToString} and reconnect them, this time getting the optimally scaled column ($y$) scores, instead.

When you are done, disconnect \textbf{OptScale} from \textbf{ObjectToString}, and save your design.

4.3.4 Weight the Scores

4.3.4.1 Connect SVD to ReWeight

The fourth step in a correspondence analysis of our data is to reweight the row and column scores according to the eigenvalues of the original SVD output. This is so that the row and column items can be plotted in the same space meaningfully, i.e., on the same scale. Thus, reweighting requires the optimally scaled scores that the \textbf{OptScale} bean just calculated, and it needs the eigenvalues (D matrix) from the \textbf{SVD} bean. So, as with \textbf{OptScale}, there will be 3 connections from 2 earlier beans.

Connect \texttt{(Tensor)SVD.getOutputD()} to \texttt{ReWeight.setInputD(Tensor)}.
Now ReWeight can get SVD’s eigenvalues when SVD is done calculating them.

4.3.4.2 Connect OptScale to ReWeight

Connect (Tensor)OptScale.getOutputX() to ReWeight.setInputX(Tensor). Now ReWeight can get OptScale’s re-scaled row-scores as soon as OptScale has calculated them.
Figure 4.39: GUIBox: **OptScale** output: Optimally scaled Row Scores

Connect *(Tensor)OptScale.getOutputY() to ReWeight.setInputY(Tensor).* Now **ReWeight** can get **OptScale**’s re-scaled column-scores as soon as **OptScale** has calculated them.
4.3.4.3 View the Weighted Values

At this point, we have finished the correspondence analysis of the original data. What we have at this point are a set of weighted row scores and a set of weighted column scores, along a certain number of dimensions.

To view these, temporarily connect up ReWeight to ObjectToString. Choose (Tensor)getOutputXw() or (Tensor)getOutputYw() as your getter method from ReWeight, to see the weighted row (Figure 4.40) or column (Figure 4.41) scores, respectively. When you are done, disconnect ReWeight from the ObjectToString.

4.3.5 Stack the Row and Column Scores

Now we have the weighted row and column scores, which can be viewed in a single space. To view them in the same picture, we are going to need to combine them into a single stacked Tensor. For this we can use another bean, StackTensors.

To be stackable, our two Tensors must have the same number of columns. Verify this by reviewing Figures 4.40 and 4.41. How many columns does each set of reweighted scores have? The number of these columns corresponds to the number of dimensions in the data.

4.3.5.1 Add StackTensors

Now, put a StackTensors bean on the Studio DesignBox, from the Socnet palette.
4.3.5.2 Connect ReWeight to StackTensors

Connect \( (\text{Tensor})\) ReWeight.getOutputXw(\) to StackTensor.setInputTensor1(\text{Tensor}). Now StackTensor can get the first (upper) \text{Tensor} from ReWeight's row scores.

Now add the column scores to the stacked \text{Tensor} you're creating: Connect \( (\text{Tensor})\) ReWeight.getOutputYw(\) to StackTensor.setInputTensor2(\text{Tensor}).
Figure 4.41: GUIBox: Weighted Column Scores

Your DesignBox (except for the ObjectToString and JTextArea beans in lower right) should approximate Figure 4.42.

4.3.5.3 View Stacked Tensor

Now StackTensors has both the input Tensors it needs to combine the weighted row and column scores into a single output Tensor. How many rows and columns
Figure 4.42: DesignBox: **StackTensors (STKT)** bean added

will there be in this output Tensor? Verify your answer by connecting (Tensor)-StackTensors.setOutputTensor to ObjectToString.setInputObject(Object), and clicking on the Calculate button again. You should see the String representation of this stacked Tensor in the JTextArea window, as in Figure 4.43. What do the rows and columns represent? Look at their labels.

Disconnect **StackTensor** from **ObjectToString** again when you are finished, and save your design.
4.4 Convert to a General Visual Representation (Graph-Array)

In this section, we use the socnet beans to convert our stacked Tensor to a generalized graph format called a GraphArray, and then from there into two different specific formats: a kinemage for viewing in the program, Mage; and a chemical format called PBD, which we will view from within Studio using the
ChemSymphony components. Both formats are String objects.

4.4.1 Convert Tensor to GraphArray

To eventually view the row and column scores in 3-D, we need to convert them from a set of matrices (Tensor) to a set of graphs (a GraphArray object).

4.4.1.1 Add TensorToGraphArray

Add a TensorToGraphArray bean to the Studio DesignBox, from the Socnet palette.

4.4.1.2 Connect StackTensors to TensorToGraphArray

Connect (Tensor)StackTensors.getOutputTensor() to TensorToGraphArray.setInputCoordinates(Tensor). Your DesignBox should now look like Figure 4.44 (except for the ObjectToString and JTextArea off to lower right).

Notice we are not connecting anything to TensorToGraphArray.setInputEdgeWeights(Tensor). This is because we have no edges (connections) for this 2-mode data. Thus, in the end, our image will appear only as balls/spheres in 3-D space, without anything (lines, pipes, edges, arcs, bonds, etc.) connecting them. We are simply interested in the relationship among the objects (categories) themselves, which we should be able to deduce from their relative proximity/distance to/from each other in 3-D space.
4.4.1.3 View GraphArray text representation

If you connect (GraphArray)TensorToGraphArray.getOutputGraphArray() to ObjectToString.setInputObject(Object), you should be able to view the resulting text: the GraphArray you’ve created, in String form, in the JTextArea window (Figure 4.45, JTextArea only shown).

4.4.1.4 Change row labels to “set” in TensorToGraphArray

However, notice that the default label for the rows is “actors”. But since SES and mental health are not actors, we would like to give these a more general

\[8\text{In the case of 2-mode data, the graphs will have no edges. However, we still use the GraphArray bean to do the translation.}\]
Figure 4.45: GUIBox/JTextArea: GraphArray as a String

description, such as “set 1” and “set 2”.

Right-click on the TensorToGraphArray icon, and choose Property (Figure 4.46). In the dialog window that pops up, click on the Property tab, and
change the `RowVertexLabel` property’s value from `actors` to `set` (Figure 4.47),

![TensorToGraphArray Properties: Changing the rowVertexLabel property](image)

Figure 4.47: **TensorToGraphArray** Properties: Changing the `rowVertexLabel` property

and click **Done**.

Now click **Calculate** again, and your **GraphArray** should appear in the **JTextArea** with the “set” labels, now, as in Figure 4.48. This will be important in the next section, when we view the data in Mage.

Disconnect **ObjectToString** when finished. Save your design.
4.5 Visualize the data with Mage

The GraphArray Socnet class is a lowest-common-denominator graph object which can be used to convert out to other graph types, such as Mage and PDB. In this section, we convert from the general representation (GraphArray), then save it to a file, then open up the external program, Mage, and view the image we’ve created.

4.5.1 Convert GraphArray to Mage

The first step in the process of visualizing the image is to convert it from the general GraphArray format into a specific format for viewing by the Mage.

Figure 4.48: GUIBox/JTextArea: GraphArray as a String, row labels corrected
4.5.1.1 Add GraphArrayToMage

Add a **GraphArrayMage** bean to the Studio DesignBox, from the **Socnet** palette.

4.5.1.2 Connect TensorToGraphArray to GraphArrayToMage

Connect *(GraphArray)TensorToGraphArray.getGraphArray()* to *GraphArrayToMage.setInputGraphArray(GraphArray)*.

4.5.1.3 Verify Mage output

Connect *(String)Mage.getOutputKinemage()* to *JTextArea.setText(String)*.

Notice that you no longer have to use **ObjectToString** before **JTextArea** to convert to a **String**, since the output of **GraphArrayToMage** is already a **String**, which is what your **JTextArea** bean is expecting as its input.

In fact, go ahead and delete **ObjectToString**, since we will not need it for the rest of the tutorial. Now your design should look like Figure 4.49).

Now click on the **Calculate** button. You should now see the first few lines of the kinemage, as a **String**, in your **JTextArea** window (Figure 4.50).

4.5.2 Save Mage output to a file

Now we have the kinemage output as a **String**. We can even view the first few lines of this string in our **JTextArea** window. But we would really like to view the image this kinemage represents. To do so, we will have to get the kinemage
4.5.2.1 Add StringToFile

Add a `StringToFile` from the `Socnet` palette to the Studio DesignBox.

4.5.2.2 Add MtSimpleFileChooser

Add a second `MtSimpleFileChooser` (from the `System` palette, I/O bean folder) to the Studio DesignBox. This will be for you to choose the name of the output file you would like to save the kinemage to.
4.5.2.3 Configure MtSimpleFileChooser component

Change the MtSimpleFileChooser’s component as in Figure 4.51 so that it says Save instead of Open, as the first MtSimpleFileChooser does. This changes the wording of the button in the lower right from Open to Save, and also changes the wording of the dialog box which comes up when you click this button. This
change in wording is simply for the benefit of the user, but is actually cosmetic only. It does not change the functionality of the MtSimpleFileChooser

4.5.2.4 Connect GraphArrayToMage to StringToFile

Connect (String)GraphArrayToMage.getOutputKinemage() to StringToFile.setInputString(String). This enables the the output kinemage to be sent to a file.

Notice you can leave GraphArrayToMage connected to JTextArea as well. This means that both the JTextArea and the StringToFile beans will get the kinemage String output.

4.5.2.5 Connect MtSimpleFileChooser to StringToFile

Connect (String)MtSimpleFileChooser.getAbsolutePath() to StringToFile.setOutputFileName(String). This way, whenever you choose a file name for your output kinemage in the MtSimpleFileChooser, it resets StringToFile to use the new file name to save the String into.

It is important to complete this step before choosing a file path name in the MtSimpleFileChooser. If you choose a file name (the next step) before connecting the beans together, the new file name will not be sent to StringToFile, even though the file name will still appear in the MtSimpleFileChooser’s com-
ponent text box.

4.5.2.6 Choose an output file name

Click on the second MtSimpleFileChooser’s Select button, and with the dialog box that pops up (Figure 4.52), navigate to the directory which contains the ses_mental_dl.txt input file. Type in ses_mental_kin.txt. This will be your output file containing the kinemage string which Mage will read in.

Now you have finished your 2-mode data design. Your Studio DesignBox should look something like Figure 4.53.
Figure 4.53: DesignBox: 2-mode dataflow network design for creating a Mage image

Click Calculate. You should now have a file named `ses_mental_kin.txt` in your `srole` directory.

Save your design.
4.5.3 View data with Mage

In this section, we finally get to view our data as a kinemage in the external program, Mage.

4.5.3.1 Start Mage

Start up your Mage program, either from the command prompt (shell), or by clicking on the icon in the directory where you installed Mage.

4.5.3.2 Open up your file

In Mage’s File menu, choose Open New File, and in the dialog window that opens up (Figure 4.54), navigate to the directory where you saved your file, choose it, and click OK.

4.5.3.3 Explore the data

Now you should see your data appear as in Figure 4.55.

Click on a few of the spheres. Notice that the blue spheres represent the parental socioeconomic strata, and that the red spheres represent the mental health categories.

Also notice that you can rotate the image in 3-D space by clicking your mouse anywhere in the window and dragging.

Find the blue sphere labelled ‘AB’. This is the sphere representing the two groups of respondents whose parents were from the highest socio-economic strata.

\[^{10}\text{If you click on the Calculate button more than once, the color assigned to the spheres may “rotate” through the legal Mage color values. This is simply a function of the Tensor-ToGraphArray bean, and does not change the data. Keep clicking the Calculate button until you get to a set of colors you like.}\]

140
If you click on the nearest red (mental health) sphere to 'AB' (just to the left of 'AB' in Figure 4.55), you will see it is labelled, 'well'. Since 'AB' and 'well' are near each other, this is the way Correspondence Analysis tells us that 'AB' and 'well' are associated. Thus, those from the higher socioeconomic strata tend to exhibit the best mental health.

Explore the rest of the data. Going down to the lower right in the figure, notice that groups 'E' and 'F' are near mental status 'impaired'. Lower mental
health is associated with people in poorer, or lower SES strata, according to the study.

Also notice that the balls are arranged almost in a flat line. This is because the CA produced only 2 dimensions — not 3 or more. Thus, the 3rd dimension is unused. The socnet beans fill in the 3rd dimension (z values) with zeros. But
the relationship between mental health and parental SES seems clear: The lower in SES your parents are, the more likely you are to suffer from poorer mental health — at least if you live in Manhattan.

Figure 4.56: Mage: Hiding the mental health ("set 1") balls

Finally, notice that "set 1" may be toggled to reveal or hide the balls representing mental health categories (Figure 4.56). The same may be done for "set 2" — the SES categories. If you want “set 1” to read “mental health” you may edit the `ses_mental_kin.txt` file manually, just by searching for the string “set...
4.6 Visualize the data with ChemSymphony

You have now seen how to create a chemical image from your data by creating a kinemage file and using the external chemical viewer, Mage, to view it. But what if you wanted to view your data with a different, all-Java viewer?

At this point, there are few freely available Java chemical viewers distributed as beans. However, one is packaged with the ChemSymphony beans you installed in Section 2.5. In this section we will use two of the ChemSymphony beans to view your data. While this viewer is not as good for labelling the spheres, it does demonstrate the flexibility of building on others’ work by using only one or two additional beans, and the ability to do everything in Java.

In this section, we will start with the design you created for viewing your data in Mage, and continue on, adding a translator to PDB format. Then we will configure the two ChemSymphony beans to show their inputs (getters) and callbacks (outputs) as red and blue nodes, respectively. Then we will add the the ChemSymphony beans, and view the data.

4.6.1 Convert GraphArray to PDB String

One of the most common formats for computer storage of chemical structures is PDB, as mentioned in Section 2.5. In this section, we will convert our previously-created GraphArray object to a PDB file instead of to a kinemage. As you may have guessed, we do not have to completely rebuild our design. All we have to do
do is add a new translator bean. In place of \texttt{GraphArrayToMage}, we will be using \texttt{GraphArrayToPDB}. We will leave the \texttt{GraphArrayToMage} bean on the DesignBox, however, so we can use it for future work.

Since many of the operations you will be doing in this section are repetitions of work you did in previous sections, I will provide illustrations only for where you’re doing something, or as a summary check at basic stopping points.

4.6.1.1 Add \texttt{GraphArrayToPDB}

From the \texttt{Socnet} palette, add a \texttt{GraphArrayToPDB} bean to your Studio DesignBox.

4.6.1.2 Disconnect \texttt{TensorToGraphArray} and \texttt{GraphArrayToMage}

Disconnect \texttt{GraphArrayToMage} from \texttt{TensorToGraphArray}.

You can leave \texttt{GraphArrayToMage} connected to \texttt{JTextArea} and \texttt{StringToFile}. Even though it would send its kinemage \texttt{String} to \texttt{JTextArea} and \texttt{StringToFile} if it calculated, \texttt{GraphArrayToMage} will not be triggered to calculate now, since it is no longer receiving \texttt{GraphArrays} from \texttt{TensorToGraphArray}.

4.6.1.3 Connect \texttt{TensorToGraphArray} to \texttt{GraphArrayToPDB}

Connect \((\texttt{GraphArray})\texttt{TensorToGraphArray.getOutputGraphArray()}) to \texttt{GraphArrayToPDB.setInputGraphArray(GraphArray)}.

Now, instead of converting a \texttt{GraphArray} to a kinemage file, we will be converting it to a PDB file.
4.6.2 Direct PDB output to JTextArea window and to a file

As we did with GraphArrayToMage, we would like to verify the output of GraphArrayToPDB. To do this, we want to see the output in our JTextArea window.

Suppose we would like to also save our PDB output to a file, for later reference, or use in other viewers, etc. Just as with Mage output, we will effect this by also sending output from GraphArrayToPDB to StringToFile.

4.6.2.1 Connect GraphArrayToPDB to JTextArea

Connect (String)GraphArrayToPDB.getOutputPDB() to JTextArea.setText(String).

4.6.2.2 Connect GraphArrayToPDB to StringToFile

We would like to save our PDB to a file, as we did with our Mage image. So connect (String)GraphArrayToPDB.getOutputPDB() to StringToFile.setInput-String(String).

4.6.2.3 Change the name of your output file

Now, let us change the name of our output file, since it is going to contain PDB data, not a kinemage. In your second MtSimpleFileChooser, choose a new file. Instead of ses_mental_kin.txt, change your file name to ses_mental_pdb.txt (Figure 4.57). Now your DesignBox should look like Figure 4.58. Save your design.
Figure 4.57: GUIBox/MtSimpleFileChooser: Changing the output filename for our PDB file

Figure 4.58: DesignBox: GraphArrayToPDB (GA2PDB) added
4.6.2.4 Test your design

Notice that at this point, we could connect **TensorToGraphArray** to either **GraphArrayToMage** or **GraphArrayToPDB**. If we disconnected one and reconnected the other, we would simply switch outputs.

At this point, we have the PDB output set up. So let us go ahead and click on the **Calculate** button, and see the output in your **JTextArea** window. It should look like Figure 4.59. This is the **String** describing our data as a molecule in the PDB chemical description text format.

![Figure 4.59: GUIBox/JTextArea: PDB output of GraphArrayToPDB](image)

This PDB **String** is also being saved to your **ses_mental_pdb.txt** file. Open that file with a text editor to verify it contains the same thing as you see in your **JTextArea** window.

---

12although the Hydrogens (H) and Carbons (C) may be Nitrogens (N) and Oxygens (O) instead. Clicking **Calculate** a second time will result in the next/alternate set of atoms being utilized by the bean.
Now we will continue on to view the PDB in a Java chemical image viewer, a bean made by a separate author, ChemSymphony. This bean is called RenderBasic.

4.6.3 Add ChemSymphony beans to design

In this section, we will add to our design two beans – RenderBasic and RenderControl – from the company, ChemSymphony, so we can view our data as a PDB chemical model in Section 4.6.4.

4.6.3.1 Add RenderBasic

Add a RenderBasic bean from the chemsymph palette. This bean should have three red input nodes/connectors. If it does not, you have not configured your ChemSymphony beans (correctly). Remove the bean, return to Section 2.5.6, follow the directions there, and then repeat this step.

4.6.3.2 Add RenderControl

Add a RenderControl bean from the chemsymph palette. This bean should have one blue output connector. If it does not, remove the bean, return to Section 2.5.6 and configure your ChemSymphony beans, and then repeat this step.

4.6.3.3 Add JTextField

Add a JTextField bean from the System palette, GUI bean folder.
4.6.3.4 Connect JTextField to RenderBasic

In order for the **RenderBasic** bean to draw the molecule in its component window, it needs to know what type of chemical format is being given to it in the **String** we send it. We can tell it what type to expect by using one of its input methods, `RenderBasic.setChemicalDataFormat(String)`. We will send it the **String**, “PDB”.

Connect `(String)JTextField.getText()` to `RenderBasic.setChemicalDataFormat(String)`.

4.6.3.5 Tell the Renderer what chemical format to expect

Now tell **RenderBasic** what chemical data format to expect. Type in ’PDB’ in the **JTextArea**, and be sure to hit the Enter or Return key on your keyboard when you are done. Hitting the Enter key is what fires the event to send the data on to **RenderBasic**.

4.6.3.6 Arrange the new bean components in the GUIBox

Arrange the three new beans in the Studio GUIBox. Since these new beans will take up considerable space, you will have to enlarge your GUIBox window, allowing it to overlap with the DesignBox on your screen. Now your GUIBox should look roughly like Figure 4.60.

4.6.3.7 Connect RenderControl to RenderBasic

The **RenderBasic** bean is ChemSymphony’s basic chemical molecular viewer. The **RenderControl** is their bean for controlling the appearance of the molecule in the **RenderBasic**. Thus, **RenderControl** needs to be connected to **Ren-
Figure 4.60: GUIBox: Re-arranging the RenderBasic and RenderControl bean components

Connect (PropertyChangeEvent)RenderControl.getIntegratedProperty- Change-

13How did I know how to set these up? I did not. Frankly, there is not adequate documentation for this older generation of ChemSymphony beans, though the corporation is reasonably accessible via email if you have a question.

I had to play with the various inputs/getters, outputs, and callbacks, until I found the ones that worked, before I could present the setup of the RenderBasic and RenderControl beans here in Section 2.5.

Also, ChemSymphony offers tutorials at http://www.chemsymphony.com/chemsymphony/Tutorials/chem_tutorials.htm. However, although the folks at ChemSymphony are well aware that their beans may be used in many different environments, the tutorials are not GeoVista Studio-based.

It would also be helpful to create a ChemSymphony palette file which could be distributed with the ChemSymphony beans so that their configuration (inputs and outputs/callbacks) were all set up when you load the beans in.

I may provide a ChemSymphony palette on the Socnet beans website (http://orion.oac.uci.edu/~jastern/socnetbeans/) at a later date. I did not for the dissertation, however, since it is important to know how to set up your own beans for Studio, as the vast majority of bean authors do not even know about Studio, much less provide a palette file for them.
Event() to RenderBasic.propertyChange(PropertyChangeEvent).

Now, whenever you make a change in the RenderControl, the RenderBasic window will be automatically updated to reflect the change.

4.6.3.8 Connect GraphArrayToPDB to RenderBasic

Here is the most important part: This is where you send the PDB String to the RenderBasic window to be interpreted and rendered as an image.

Connect (String)GraphArrayToPDB.getOutputPDB() to RenderBasic.setChemicalContents(String).

Your design should now look something like Figure 4.61. Save the design.

4.6.4 View data with ChemSymphony

In this section, we finally get to view our data as a PDB chemical model using the beans we added in Section 4.6.3.

4.6.4.1 Run the design

As mentioned in Section 4.6.3.5, make sure you have pressed the Enter key in the JTextArea window containing the word, ‘PDB’.

Now click on the Calculate button. You should see something like Figure 4.62 in the RenderBasic window (though I’ve removed the black background). If you see grey lines instead of red and blue, just click Calculate again. Your JTextArea’s PDB data should show Nitrogens (N) and Oxygens (O), not Hydrogens (H) and Carbons (C).
4.6.4.2 Change the view using RenderControl

This is just a wireframe view of the data which the RenderBasic renderer defaults to when it first runs. We have to change a few things about the way it draws the data, in order to see it better.

In your RenderControl’s component in the GUIBox, make sure the General tab is selected at the top. Now change DisplayStyle to Balls, as in Figure 4.63.
Figure 4.62: ChemSymphony **RenderBasic**: Srole SES-Mental data viewed for first time

![Image of Display Style settings]

Figure 4.63: ChemSymphony **RenderControl**: Changing the **Display Style** to **Balls**
You should notice an immediate change in the picture to something like Figure 4.64.

Now we will make a few more changes. Each time you make a selection choice in the RenderControl component, you should see an immediate change in the data RenderBasic window.

In the RenderControl, change AtomTypes from Empty to Index. You should now see the atoms numbered.

Now change Rendering quality from Normal to Highest.

We need to make the atoms appear smaller. Click on the Chemistry tab, and reduce Atomic radius factor from 50% to 10%, as seen in Figure 4.65. Now your
Figure 4.65: ChemSymphony **RenderControl**: Reducing the size of the rendered atoms

**RenderBasic** render window should show something a little bit more friendly, as that seen in Figure [4.66](#). Studio’s full GUIBox with your data running should look roughly like Figure [4.67](#).

Since the PDB data format itself does not allow for individual labelling of atoms, so we’ve lost the convenient numbering of the rows and columns of the original data. However, we can refer back to Section [4.3.5](#) (which starts on page [124](#)). In that section, we stacked our row and column scores into a single **Tensor**, resulting in the output seen in Figure [4.43](#) (p. [128](#)). There we can see from the **rowlabels** line that the first three rows of this stacked **Tensor** are the health categories, and the last four are the socio-economic strata. Mapping these one-to-one onto the PDB atoms in Figure [4.59](#) (p. [148](#)), we can see that
the Oxygen (O) atoms are the mental health categories. These appear as red spheres. Concomitantly, the SES strata appear as the last four atoms, Nitrogen (N) atoms appearing blue in the RenderBasic component window.

Since we chose an index labelling style, above, we see the numbers labelling the atoms. The Oxygen atom numbered with index '0' (left-most in Figure 4.66) corresponds to the 'well' mental health category. Nitrogen atom '3' corresponds to the SES category, 'AB' – the highest strata. Furthermore, the lowest SES strata – blue Nitrogen atoms 5 and 6, or groups 'E' and 'F' – appear nearest to, and are associated with impaired mental health (red Oxygen atom 2).

This is just what we found in Section 4.5 (p. 133) when viewing the data in Mage, though because of the superior labelling facilities of that program it was easier to identify the rows of the original data.

However, just as with Mage, the ChemSymphony RenderBasic component
Figure 4.67: GUIBox: Final 2 mode data viewer running SRole SES-Mental data as PDB with ChemSymphony beans

allows you to use your mouse to explore, getting a better feel for your data by rotating the molecule in three dimensions. Go ahead and do that now. Also, play with some of the **RenderControl** controls and see how they change the image you view.

4.7 Conclusion

This concludes the tutorial for creating a view of 2-mode data with the socnet beans, Mage, and ChemSymphony. Each of the viewers has advantages and disadvantages. ChemSymphony allows you to view the data within the Java application itself. It also has the capability to label the atoms with their "atom
number”s – a convenient feature for social network researchers. Mage on the other hand has many viewing advantages. One of the most important is that it allows you to view more than one frame – as we will see in the chapter when we will be looking at how to create views of 1-mode data over time.
CHAPTER 5

Tutorial Part 2: Correspondence Analysis of One-Mode data with the Socnet Beans

5.1 Introduction

This tutorial assumes that you have completed the 2-mode example tutorial in the previous chapter. Thus, you have Geovista Studio running, and you have the design, 2modeviewer.gvd, opened and running. You also have Mage running.

In this section, we will continue with the design you made, altering it to view a 1-mode data set. As with the 2-mode example, the design will create at the end an image which can be viewed in two separate molecular viewers: Mage and ChemSymphony.

However, there will be some basic differences in the image you create, because this is 1-mode data. First, because the rows and columns represent the same entities, there is only one set of spheres. Second, the spheres will be connected by lines, or edges.

The dataset we will use comes from Newcomb (1961). It contains 15 weeks of rankings. Seventeen previously unacquainted college men at the University of Michigan were recruited in the mid 1950’s to spend a semester in housing that was paid for by the study. Each week each man was to provide a ranking of his affinity to the other men, from 1 to 17. Thus there are 15 17x17 matrices. The
data are non-symmetric.

In this tutorial we are interested in learning to create a 1-mode data flow design in Studio. Because this data is taken over a period of weeks, however, we will be able to create not just one image, but a series of images, one from each week. This will allow us to “animate” the final output and answer the simple question, “Do the relationships among these actors settle down into a more or less similar pattern in the space of 16 weeks?”

5.2 Setup

We are going to be making some important changes to the 2-mode design to turn it into a 1-mode design. We will set ourselves up for these changes, now.

5.2.1 Create a new directory

As we did in Section 4.2.1 create a folder/directory for working with your data. We will call this directory, newcomb. Put it wherever you like.

5.2.2 Rename the design and save it to the new directory

Since we will be working with an entirely different type of dataset which will require a modification to the design, let us save our design under a different name.

Right now you should still have your 2 mode design (2modeviewer.gvd) up and running in Studio. If not, load it now.

Now save it under a different name, 1modeviewer.gvd, and instead of saving it to the srole directory, save it to the newly-created newcomb directory, as in
5.2.3 Disconnect Correspondence Analysis from the design

We will continue using the correspondence analysis (CA) beans, SqRtNorm, SVD, OptScale, and ReWeight. However, let us disconnect that part of the design now, so that it is not sent any data.

5.2.3.1 Disconnect ReadDL from SqRtNorm

Disconnect ReadDL from SqRtNorm.
5.2.3.2 Disconnect ReadDL from OptScale

Also, disconnect ReadDL from OptScale, as well.

Now we are ready to read in our input file, without worry that it will get automatically sent into the correspondence analysis, yet.

5.2.3.3 Remove StackTensors

Since this is one mode data, the rows and the columns represent the same entities (the men in the Newcomb fraternity). Therefore we will not have to stack two different sets of entities after the correspondence analysis is finished.

Delete the StackTensors bean by right-clicking on STKT, and choosing Edit, then Delete, as seen in Figure 5.2 in before and after shots. This will also automatically disconnect StackTensors from ReWeight and TensorToGraphArray.

5.2.4 Rearrange the design

In this tutorial, we will be adding a few more beans. This will crowd Studio’s DesignBox unless we move things around a bit.
5.2.4.1 Split design into three sections

In order to keep things less crowded, let us open up our design. We are going to be adding in three more beans before the correspondence analysis (right after ReadDL), and a fourth new bean after the correspondence analysis (immediately before TensorToGraphArray).

I have split my DesignBox into three sections, roughly corresponding to: 1) reading in the data, 2) correspondence analysis 3) creating the output image. You can see this in Figure 5.3.

Of course, you do not have to do it like I did. You can spread out your icons horizontally, if you like, and make one long chain. It does not really matter. The only thing that matters in the end is what is connected to what.

Save your design.

5.3 Read in the Data

Now we will adapt our design so that it reads in the 1-mode Newcomb dataset.

5.3.1 Create the input file

First, create a text file in your newcom directory, named newfrat_d1.txt. The first few lines (header and first matrix) of this large file may be seen in Figure 5.4. This is not the complete file.

This data has been provided with the last several versions of UCINET, so if you have a copy of UCINET, you may create newfrat_d1.txt by exporting a DL file from UCINET’s included data file, NEWFRAT.###H, or you may download newfrat_d1.txt from the socnet beans website, http://orion.oac.uci.edu/
Figure 5.3: DesignBox: Beans rearranged to make room for new beans
```plaintext
DL
N=17  NM=15
FORMAT = FULLMATRIX DIAGONAL PRESENT
LEVEL LABELS:
NEWC0
NEWC1
NEWC2
NEWC3
NEWC4
NEWC5
NEWC6
NEWC7
NEWC8
NEWC10
NEWC11
NEWC12
NEWC13
NEWC14
NEWC15
DATA:
   0  7 12 11 10 4 13 14 15 16 3  9  1  5  8  6  2
   8  0 16  1 11 12 2 14 10 13 15 6  7  9  5  3  4
  13 10  0  7  8 11  9 15  6  5  2  1 16 12  4 14  3
  13  1 15  0 14  4  3 16 12  7  6  9  8 11 10  5  2
  14 10 11  7  0 16 12  4  5  6  2  3 13 15  8  9  1
   7 13 11  3 15  0 10  2  4 16 14  5  1 12  9  8  6
  15  4 11  3 16  8  0  6  9 10  5  2 14 12  13  7  1
   9  8 16  7 10  1 14  0 11  3  2  5  4 15 12 13  6
  16  8 14 13 11  4 15  0  7  1  2  9  5 12 10  3
  16  9 14 11  4  3 10  7  0 15  8 12 13  1  6  5
 12  7  4  8  6 14  9 16  3 13  0  2 10 15 11  5  1
 15 11  2  6  5 14  7 13 10  4  3  0 16  8  9 12  1
   1 15 16  7  4  2 12 14 13  8  6 11  0 10  3  9  5
 14  5  8  6 13  9  2 16  1  3 12  7 15  0  4 11 10
 16  9  4  8  1 13 11 12  6  2  3  5 10 15  0 14  7
  8 11 15  3 13 16 14 12  1  9  2  6 10  7  5  0  4
  9 15 10  2  4 11  5 12  3  7  8  1  6 16 14 13  0
...
```

Figure 5.4: Beginning of DL-formatted Newcomb fraternity data
Save it to your `newcomb` directory — the same place you saved your `1modeviewer.gvd` design.

5.3.2 Read in the the input file

5.3.2.1 Select file

As in section 4.2.3.8, click on the `MtSimpleFileChooser`'s visual component’s Select button, navigate to the directory where you saved your `newfrat_dl.txt` file, and choose it. Now you should have the file name showing in the component,

![File Name: /home/jastern/dissert/dev/testfiles/newcomb/newfrat_d1.txt](image)

Figure 5.5: DesignBox/MtSimpleFileChooser: Input file chosen for Newcomb fraternity data

as in Figure 5.5

5.3.3 Verify data

We want to view the input Tensor we just read in. The easiest way to do that is to view it as a String in the JTextArea window.

5.3.3.1 Add ObjectToString

To view the data in the JTextArea, we need to first convert the Tensor to a String. To do this, we will need to replace the ObjectToString bean we removed in Section 4.5.1.3.
Put a new **ObjectToString** bean back onto the Studio DesignBox.

### 5.3.3.2 Connect ObjectToString to JTextArea

Now, just as we did in Section 5.3.3.2, connect `(String)ObjectToString.GetString()` to `JTextArea.setText(String)`.

Now any **String** that **ObjectToString** produces should be visible in the **JTextArea** window.

### 5.3.3.3 Connect ReadDL to ObjectToString

Connect `(DL)ReadDL.getOutputTensor()` to `ObjectToString.setInputObject(Object)`.

Remember, since a **Tensor** is an **Object**, we can connect these two connectors. Now your design should look something like Figure 5.6.

### 5.3.3.4 Run the design

Click on the **Calculate** button in the Studio GUIBox. You should now see the first few lines of the **Tensor** we created when reading in the DL file, as in Figure 5.7.

This verifies you have read in the DL file, and that it contains the Newcomb data, in the form of a **Tensor**. Notice that Figure 5.7 looks different than Figure 5.4 because Figure 5.4 is in DataLanguage format and Figure 5.7 is output as a **Tensor**.

---

1. *not (DL)ReadDL.getOutputDL()*

2. *which you would still see in the JTextArea if you had chosen ReadDL.getOutputDL()* instead of `ReadDL.getOutputTensor()` in Section 5.3.3.3. In that case you would have received a **DL** object and when a **DL** prints itself out, it will print out in DataLanguage format. But a **Tensor** prints out in a different format.
Figure 5.6: DesignBox: **ObjectToString** added
5.3.3.5 Disconnect ReadDL from ObjectToString

Disconnect ReadDL from ObjectToString when you are done verifying.

Save your design.

5.4 Reverse the data

You may have noticed that there are a few differences between the Newcomb and Srole datasets. Not only are the Newcomb data 1-mode, but it is also ranked data: Each house member ranks all the others weekly in terms of his proximity or affinity to the other members. But because it is ranked data, the numbers in the cells get larger the further away alter is from ego. This is a type of “distance” data: Each ranker gives the person he is closest with a ‘1’, the next a ‘2’, all the way down to the 17th member. Also, as you can see in Figure 5.7, diagonal entries get a ‘0’, meaning that each ranker is 0 distance from (i.e., closest to)
himself.

However, to perform an SVD on this data, we will need to be working with proximity, or closeness data, not distance data. That means that the cell values will have to increase with proximity, not decrease.

The easiest way to do this is to simply reverse the input Tensor, exchanging the largest values for the smallest. Thus, a 17 will become a 0, a 0 a 17. A 16 will become a 1, and visa versa. And so on.

5.4.1 Reverse the data with a ReverseTensor bean

We need to reverse the Tensor after reading it in from the file, but before it goes into the SVD. Therefore, we need to insert the ReverseTensor bean right after the ReadDL bean.

5.4.1.1 Add ReverseTensor bean

From the Socnet palette, jastern bean folder, put a ReverseTensor bean onto the DesignBox.

5.4.1.2 Connect ReadDL to ReverseTensor

Now connect (DL)ReadDL.getOutputTensor() to ReverseTensor.setInputTensor(Tensor). The Tensor output from ReverseTensor will now contain closeness data. That is, the highest number (in this case, 17) will be reserved for self/ego, the next highest for the closest affine, or friend. and so on, down to the person farthest away from ego, which will have a 0.
5.4.2 Verify the Reversal

We can verify that the data now contain closeness (or proximity) data as we did before: by converting the (reversed) Tensor to a String and sending the String to the JTextArea window for viewing.

5.4.2.1 Connect ReverseTensor to ObjectToString

Connect (Tensor) ReverseTensor.getOutputTensor() to ObjectToString.setInputObject(Object). The top part of your Studio DesignBox should now look some-

![DesignBox: ReverseTensor added](image)

Figure 5.8: DesignBox: ReverseTensor added

thing like Figure 5.8. (Note I am only showing the top part of the DesignBox, and the window borders have been removed, as well.)

5.4.2.2 Click Calculate

Click Calculate. The JTextArea should contain the first few lines of the output of the reversed Tensor, as in Figure 5.9. You can verify that the diagonal now
Figure 5.9: GUIBox/JTextArea: the Reversed Newfrat Tensor

contains the highest value in each row, not the lowest.

5.4.2.3 Disconnect ReverseTensor from ObjectToString

Disconnect ReverseTensor from ObjectToString when you are satisfied that your data have been reversed.

Save your design.

5.5 Symmetrize the data

Now we have reversed the data, so that they represent proximities, not distances. However, there is another problem we need to solve before we can send the data through the correspondence analysis. The problem is that the data are not symmetric: Person A may not rank person B in exactly the same order as B ranks A. In fact, it is usually the case that they do not. For instance, in Figure 5.9.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16.0</td>
<td>5.0</td>
<td>9.0</td>
<td>3.0</td>
<td>0.0</td>
<td>2.0</td>
<td>6.0</td>
<td>0.0</td>
<td>4.0</td>
<td>8.0</td>
<td>0.0</td>
<td>10.0</td>
<td>4.0</td>
<td>0.0</td>
<td>13.0</td>
<td>7.0</td>
<td>15.0</td>
</tr>
<tr>
<td>2</td>
<td>8.0</td>
<td>0.0</td>
<td>0.0</td>
<td>15.0</td>
<td>5.0</td>
<td>4.0</td>
<td>2.0</td>
<td>6.0</td>
<td>3.0</td>
<td>1.0</td>
<td>10.0</td>
<td>9.0</td>
<td>7.0</td>
<td>11.0</td>
<td>15.0</td>
<td>0.0</td>
<td>13.0</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>15.0</td>
<td>0.0</td>
<td>1.0</td>
<td>16.0</td>
<td>0.0</td>
<td>7.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>10.0</td>
<td>14.0</td>
<td>0.0</td>
<td>15.0</td>
<td>0.0</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td>4</td>
<td>3.0</td>
<td>5.0</td>
<td>0.0</td>
<td>12.0</td>
<td>10.0</td>
<td>10.0</td>
<td>0.0</td>
<td>1.0</td>
<td>4.0</td>
<td>0.0</td>
<td>2.0</td>
<td>11.0</td>
<td>15.0</td>
<td>0.0</td>
<td>0.0</td>
<td>4.0</td>
<td>12.0</td>
</tr>
<tr>
<td>5</td>
<td>2.0</td>
<td>6.0</td>
<td>0.0</td>
<td>0.0</td>
<td>4.0</td>
<td>12.0</td>
<td>11.0</td>
<td>10.0</td>
<td>14.0</td>
<td>13.0</td>
<td>0.0</td>
<td>1.0</td>
<td>8.0</td>
<td>7.0</td>
<td>15.0</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>9.0</td>
<td>3.0</td>
<td>0.0</td>
<td>0.0</td>
<td>13.0</td>
<td>1.0</td>
<td>9.0</td>
<td>14.0</td>
<td>12.0</td>
<td>0.0</td>
<td>2.0</td>
<td>11.0</td>
<td>15.0</td>
<td>0.0</td>
<td>0.0</td>
<td>7.0</td>
<td>8.0</td>
</tr>
<tr>
<td>7</td>
<td>1.0</td>
<td>12.0</td>
<td>0.0</td>
<td>3.0</td>
<td>15.0</td>
<td>0.0</td>
<td>16.0</td>
<td>0.0</td>
<td>14.0</td>
<td>12.0</td>
<td>0.0</td>
<td>2.0</td>
<td>11.0</td>
<td>15.0</td>
<td>2.0</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td>8</td>
<td>7.0</td>
<td>8.0</td>
<td>0.0</td>
<td>0.0</td>
<td>9.0</td>
<td>6.0</td>
<td>15.0</td>
<td>0.0</td>
<td>16.0</td>
<td>5.0</td>
<td>3.0</td>
<td>0.0</td>
<td>11.0</td>
<td>15.0</td>
<td>0.0</td>
<td>4.0</td>
<td>3.0</td>
</tr>
<tr>
<td>9</td>
<td>10.0</td>
<td>0.0</td>
<td>0.0</td>
<td>2.0</td>
<td>3.0</td>
<td>5.0</td>
<td>12.0</td>
<td>1.0</td>
<td>16.0</td>
<td>0.0</td>
<td>9.0</td>
<td>4.0</td>
<td>0.0</td>
<td>11.0</td>
<td>4.0</td>
<td>6.0</td>
<td>13.0</td>
</tr>
</tbody>
</table>
you can see that person 2 ranks person 1 at 8.0 (row 2, column 1), while person 1 ranks person 2 at 9.0 (row 1, column 2).

Therefore, a correspondence analysis of the rows would yield a different picture than that of the columns. And since we are simply interested in overall proximity and whether it settles into a pattern, we would like these to agree.

But how to do so? The way is to symmetrize the data in some way: to make the ranks agree. We will do it by taking the sum of the two values. Thus, in the above example, both values will equal 17.

5.5.1 Symmetrize the data with a SymmetrizeTensor bean

We need to symmetrize the data after it has been reversed, but before it goes into CA. So we will insert the SymmetrizeTensor bean right after the ReverseTensor bean.

5.5.1.1 Add SymmetrizeTensor bean

From the Socnet palette, jastern bean folder, put a SymmetrizeTensor bean onto the Studio DesignBox.

5.5.1.2 Connect ReverseTensor to SymmetrizeTensor

Now connect (Tensor)ReverseTensor.getOutputTensor() to SymmetrizeTensor.setInputTensor(Tensor). The Tensor output from ReverseTensor will now contain closeness data. That is, the highest number (in this case, 17) will be reserved for self/ego, the next highest for the closest affine, or friend. and so on, down to the person farthest away from ego, which will have a 0.

[^3]: via an inversion of the matrix
5.5.1.3 Choose symmetrization by SUM

Right-click on the **SymmetrizeTensor** bean icon in the DesignBox, and choose **Property**. Then, when the dialog box opens, click on the **Properties** tab and make sure that the **Method** property’s value is at **SUM**, as seen in Figure 5.10. With other datasets you use in the future, you may decide to use the **MAXIMUM** value of the row/column values, or one of the other methods, as you feel is appropriate for your data. For now, we will use **SUM**. Click **Done**.

![Properties for SymmetrizeTensor](image)

Figure 5.10: Properties for **SymmetrizeTensor**: checking method is by **SUM**
5.5.2 Verify the Symmetry

As before, we will verify that the data now contain symmetrized data by using ObjectToString and JTextArea.

5.5.2.1 Connect SymmetrizeTensor to ObjectToString

Figure 5.11: DesignBox: SymmetrizeTensor added

Connect (Tensor)SymmetrizeTensor.getOutputTensor() to ObjectToString. setInputObject(Object). The top part of your Studio DesignBox should now look something like Figure 5.11.

5.5.2.2 Click Calculate

Click Calculate. The JTextArea should contain the first few lines of the output of the symmetrized Tensor, as in Figure 5.12. You can verify that values around the diagonal are now equal to each other. For instance, whereas they used to be 8
Figure 5.12: GUIBox/JTextArea: the symmetrized newfrat Tensor

and 9, respectively, person 2 (row 2) now gives person 1 (column 1) a (summed) rank of 17, and visa versa. Also, all persons give themselves a summed rank of 32 now, instead of 16.

5.5.2.3 Disconnect SymmetrizeTensor from ObjectToString

Disconnect **SymmetrizeTensor** from **ObjectToString** when you are satisfied that your data have been symmetrized.

Save your design.

5.6 Collapse the data

Having reversed the data, so that they represent proximities, and symmetrized them, so that rankers agree, there is one more pre-CA transformation we should perform.
We could go ahead connect up the CA part of the dataflow design at this point, and the dataflow would work. However, there would be a problem: Since the newfrat data are time-sequenced, we have essentially 15 different data sets. Using Mage or any other viewer to sequence through them, we will effect a sort of “movie”. However, any movie created at this point would be created by doing a CA on each of the separate “frames”, or matrices, in the Tensor. And there is a subtle problem here: Since a CA is done on each separate matrix, then each matrix will be scaled only according to its own data. Thus, the size of the scales will vary from frame to frame. So, when we sequence through the frames, they will appear to jump around, magnify, and de-magnify. This will make it harder for the human eye to detect movement of each sphere vis-à-vis the other spheres.

The easiest way to solve this is to simply perform a CA on all of these 15 matrices as if they were one large matrix. This puts them all into the same scale. Then, if we could break them back apart again, we could view them as 15 separate frames.

The way to do this is to stack the matrices of the newfrat Tensor into a single large matrix, perform the CA, and then expand the matrix of that Tensor back out.

5.6.1 Collapse the data with a CollapseTensor bean

We need to collapse the data after it has been symmetrized, but before it goes into CA. So we will insert the CollapseTensor bean right after the SymmetrizeTensor bean.
5.6.1.1 Add CollapseTensor bean

From the Socnet palette, jastern bean folder, put a CollapseTensor bean onto the Studio DesignBox.

5.6.1.2 Connect SymmetrizeTensor to CollapseTensor

Now connect (Tensor)SymmetrizeTensor.getOutputTensor() to CollapseTensor.setInputTensor(Tensor). After we run this design, the Tensor produced by CollapseTensor will contain a single matrix, instead of 15 matrices.

5.6.2 Verify the matrices have been collapsed

Now we will check the data are now collapsed, with ObjectToString, as before.

5.6.2.1 Connect CollapseTensor to ObjectToString

Connect (Tensor)CollapseTensor.getOutputTensor() to ObjectToString.setIn-
putObject(Object). The top part of your Studio DesignBox should now look something like Figure 5.13.

5.6.2.2 Click Calculate

Click Calculate. The JTextArea should contain the first few lines of the output

```
title = "tns0, reversed, symmetrized by SUM, collapsed vertically"
nummats = 1
numrows = 255
numcols = 17
matrixlabels = "mat1"
rowlabels = "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13" "14" "15" "16" "17" "1" "2"
collabels = "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13" "14" "15" "16" "17"
data =
32.0 17.0 7.0 8.0 8.0 21.0 4.0 9.0 11.0 14.0 17.0 8.0 30.0 13.0 8.0 18.0 21.0
17.0 32.0 6.0 30.0 11.0 7.0 26.0 10.0 6.0 3.0 10.0 15.0 10.0 18.0 18.0 18.0 13.0
7.0 6.0 32.0 10.0 13.0 10.0 12.0 1.0 18.0 18.0 26.0 29.0 0.0 12.0 24.0 3.0 19.0
8.0 30.0 10.0 32.0 11.0 25.0 26.0 9.0 6.0 11.0 18.0 17.0 17.0 15.0 14.0 24.0 28.0
8.0 11.0 13.0 11.0 32.0 1.0 4.0 18.0 14.0 15.0 24.0 24.0 15.0 4.0 23.0 10.0 27.0
21.0 7.0 10.0 25.0 1.0 32.0 14.0 29.0 17.0 12.0 4.0 13.0 29.0 11.0 10.0 8.0 15.0
4.0 26.0 12.0 26.0 4.0 14.0 32.0 12.0 19.0 19.0 18.0 23.0 6.0 18.0 8.0 11.0 26.0
9.0 10.0 1.0 9.0 18.0 29.0 12.0 32.0 6.0 19.0 14.0 14.0 14.0 1.0 8.0 7.0 14.0
11.0 6.0 18.0 6.0 14.0 17.0 19.0 6.0 32.0 18.0 28.0 20.0 10.0 26.0 14.0 21.0 26.0
```

Figure 5.14: GUIBox/JTextArea: the collapsed newfrat Tensor

of the symmetrized Tensor, as in Figure 5.14. You can verify that there is now 1 matrix, instead of 15, and that the number of rows is now 255 (Why is this?). The first few cell values should also appear exactly the same as before.

5.6.2.3 Disconnect CollapseTensor from ObjectToString

Disconnect CollapseTensor from ObjectToString when you are satisfied that your data have been collapsed.

Save your design.
5.7 Reconnect Correspondence Analysis beans

Now we are ready to perform a CA on our reversed, symmetrized, collapsed Tensor. So we will connect the CA beans back in.

5.7.1 Move ObjectToString bean

Let us move the ObjectToString bean down, closer to the JTextArea, since we will not be connecting it to anything from the top, anymore.

5.7.2 Reconnect CA beans

Now let us connect up the Correspondence Analysis, again.

5.7.2.1 Connect CollapseTensor to SqRtNorm

As you may recall from Section 4.3.1.1, SqRtNorm is the pre-transformation bean we use before SVD to normalize the data. Thus it is the “first” bean in the CA part of the design – the bean we will want to send our reversed, symmetrized, collapsed Tensor.

Connect (Tensor)CollapseTensor.getOutputTensor() to SqRtNorm.setInputTensor(Tensor).

5.7.2.2 Connect CollapseTensor to OptScale

As with the 2-mode data, OptScale also needs the same “original” data matrix that ReWeight needs. So let us connect it, as well.

Connect (Tensor)CollapseTensor.getOutputTensor() to OptScale.setInputO-(Tensor).
5.7.3 Verify the Correspondence Analysis of the Collapsed Tensor

Let us check the collapsed data are running through the CA.

5.7.3.1 Connect ReWeight to ObjectToString

The last bean in the Correspondence Analysis is the post-transformation bean, ReWeight. To verify the Correspondence Analysis is working properly, let us look at the row scores which ReWeight outputs. Connect (Tensor)ReWeight.getOutputXw() to ObjectToString.setInputObject(Object). Your Studio DesignBox should now approximate Figure 5.15.

5.7.3.2 Click Calculate

Click Calculate. The JTextArea should contain the first few lines of the row scores of the newfrat Tensor, as in Figure 5.16. All the values should vary from −1 to +1.

5.7.3.3 Disconnect ReWeight from ObjectToString

Disconnect ReWeight from ObjectToString when you are satisfied that your correspondence analysis is working properly.

Save your design.

5.8 Expand the data

Now that the CA outputs a Tensor containing a stacked matrix of row scores, we need to expand this matrix back out to 15 matrices containing the row scores corresponding to the 15 original matrices — one for each week.
Figure 5.15: DesignBox: Correspondence Analysis beans reconnected
Figure 5.16: **GUIBox/JTextArea**: the row scores of the collapsed `newfrat` Tensor

### 5.8.1 Expand the data with the ExpandTensor bean

The `ExpandTensor` bean will expand the matrices back out.

`ExpandTensor` requires two inputs. First, of course, it needs the `Tensor` to be expanded.

Second, it needs to know how many matrices to expand back out to. How does it get this information? In one of two ways. One way is that you can provide it with a copy of the original matrix labels from the (pre-collapsed) data. From the number of labels provided, `ExpandTensor` will deduce the number of matrices to create, and the number of rows per matrix. The second way to give it the number of matrices is to set its `NumberOfNewMatrices` property. In this second case, `ExpandTensor` will have to create its own matrix labels. These two methods are mutually exclusive: If you use the second method, then any provided matrix labels will be ignored. If you provide matrix labels, then any
number you give to the NumberOfNewMatrices property will be ignored.

You have to tell ExpandTensor which method you are going to use, by setting the RequireMatrixLabels boolean property to true (for the first method) or false (for the second).

The first method is preferred, since it provides more specific matrix labels representing the original data. Otherwise, with the second method, ExpandTensor will just use generic matrix labels, such as “mat1”, “mat2”, etc.

Fortunately we can obtain the original matrix labels from the CollapseTensor bean, so we will use that method to give ExpandTensor the number of matrices and their labels.

5.8.1.1 Add ExpandTensor bean

From the Socnet palette, jastern bean folder, put a ExpandTensor bean onto the Studio DesignBox.

5.8.1.2 Connect ReWeight to ExpandTensor

Let us give ExpandTensor the collapsed/combined row scores from ReWeight. Connect (Tensor)ReWeight.getOutputXw() to ExpandTensor.setInputTensor(Tensor).

5.8.1.3 Set RequireMatrixLabels property to true

Now let us tell the ExpandTensor bean which way we will be communicating the number of matrices: via method 1: providing the matrix labels themselves.

Right-click on the ExpandTensor icon in the DesignBox and click on Property. In the dialog window that pops up (Figure 5.17), click on the Properties
Figure 5.17: Properties for **ExpandTensor**: requiring Matrix Labels

...tab, and verify that the `requireMatrixLabels` property is set to `True`. If it is not, click it. Click *Done*.

### 5.8.1.4 Connect CollapseTensor to ExpandTensor

Now let us give **ExpandTensor** the original matrix labels for the rows of the reversed, symmetrized data, before it was collapsed.

Connect `(StringList)CollapseTensor.getInputTensorMatrixLabels()` to **Exp-
pandTensor.setNewMatrixLabels(StringList).

5.8.2 Verify the Tensor’s matrix has been expanded

Let us check the Tensor’s single matrix of 255 rows of row-scores has been expanded back out to 15 matrices of 17 rows each of row-scores.

5.8.2.1 Connect ExpandTensor to ObjectToString

Connect (Tensor)ExpandTensor.getOutputTensor() to ObjectToString.setInputObject(Object). Your Studio DesignBox should now approximate Figure 5.18.

5.8.2.2 Click Calculate

Click Calculate. The JTextArea should contain the first few lines of the row scores of the expanded newfrat Tensor, as in Figure 5.19. The cell values should be identical to those in Figure 5.16 except that there should now be 15 matrices and 17 rows per matrix.

5.8.2.3 Disconnect ExpandTensor from ObjectToString

Disconnect ExpandTensor from ObjectToString when you are satisfied that your Tensor’s matrices have been expanded out.

Save your design.
Figure 5.18: DesignBox: **ExpandTensor** added
Figure 5.19: GUIBox/JTextArea: the row scores of the expanded `newfrat` Tensor

### 5.9 Convert to a Generalized Graph Representation

Now we have the row scores for the 17 men in each of the 10 separate matrices, and these are held in a single Tensor. We want to convert this Tensor into a GraphArray and from there to the two independent graphic formats which we have used so far: a kinemage file which may be viewed in Mage, and a PDB file which may viewed with the ChemSymphony molecular renderer.

The first step in this process is to connect the ExpandTensor bean into the third section of our design — the section which converts to a GraphArray and then out to a Mage or ChemSymphony molecular image file.

---

4and since the matrices were symmetric, these would equal the column scores, if we calculated them
5.9.1 Provide TensorToGraphArray with Coordinates

The GraphArray we create will have two differences from the one we created in Chapter 4. First, it will only have one set of spheres. These spheres will represent both the rows and the columns, since both the rows and the columns represent the same set of actors: the men in Newcomb’s fraternity. Thus, in any given picture, all the spheres should come out having the same color. Second, there will be lines connecting the spheres. These lines will represent rankings, or weights, of a certain cutoff value or higher.

Therefore, we need to provide TensorToGraphArray with not only the coordinates for the spheres (as we did with 2-mode data), but also the weights of the lines connecting them.

5.9.1.1 Set TensorToGraphArray’s IncludeWeights property to true

Because we are going to be sending TensorToGraphArray both the sphere coordinates and the edge weights, we need some way to tell TensorToGraphArray to wait for both inputs before it creates its output GraphArray. The way to do this is to set TensorToGraphArray’s IncludeWeights property to true.

To do this, right-click on the icon for TensorToGraphArray in Studio’s DesignBox. Choose Property. Then, in the dialog box that pops up (Figure 5.20), make sure that the value for IncludeWeights is set to true.

Do not click on the Done button yet.

5.9.1.2 Set TensorToGraphArray’s RowVertexLabels property

While we are here, we should also change the labelling of our spheres. Since our 1-mode data’s objects are no longer sets of things, but are people, let us change
Figure 5.20: **TensorToGraphArray** properties: Setting `includeWeights` to `true`

the labelling for our sets of balls which will later appear in our Mage output.

Change the value for the `RowVertexLabel` property from `set` to `actors`, as in Figure 5.21.

Click **Done** to close the dialog box.
Figure 5.21: **TensorToGraphArray** properties: Setting `rowVertexLabel` to `actors`

### 5.9.1.3 Connect ExpandTensor to TensorToGraphArray

Connect `(Tensor)ExpandTensor.getOutputTensor()` to `TensorToGraphArray.setInputCoordinates(Tensor)`. This will allow **ExpandTensor** to provide **TensorToGraphArray** with the coordinates it needs to situate the spheres in 3-dimensional space.
5.9.2 Provide TensorToGraphArray with Edge Weights

Okay, now we need to provide the TensorToGraphArray with edge weights. Where can we get these?

We need to go back to before the CA was performed. We can use the reversed, symmetrized (but not collapsed) data, as the edge weights. These cell values represent how strongly connected each fraternity member is to each other.

5.9.2.1 Connect SymmetrizeTensor to TensorToGraphArray

Connect (Tensor)SymmetrizeTensor.getOutputTensor() to TensorToGraphArray.setInputEdgeWeights(Tensor). This will provide TensorToGraphArray with the needed edge weights.

5.9.3 Verify the TensorToGraphArray is working

Let us verify that the network up to TensorToGraphArray is working.

5.9.3.1 Disconnect TensorToGraphArray from GraphArrayToPDB

In Chapter 4 we left TensorToGraphArray connected to GraphArrayToPDB. Let us disconnect them now, as we do not want the final section of the design to work just yet.

5.9.3.2 Connect TensorToGraphArray to ObjectToString

Connect (GraphArray)TensorToGraphArray.getOutputGraphArray() to ObjectToString.setInputObject(Object). Your Studio DesignBox should now approximate Figure 5.22. Note that because the design has gotten a bit complex, you
Figure 5.22: DesignBox: TensorToGraphArray added
may have to “fiddle” and move some of the icons around slightly in order to be able to discern the connecting lines correctly.

5.9.3.3 Click Calculate

Click Calculate. The JTextArea should contain the first few lines of the GraphArray:

```
GraphArray Title: GraphArray of tns0, reversed, symmetrized by SUM
GraphArray Descr.
IGraph Title: NEWCO
GGraph Descr.:
Nodes:
N 1
  group: actors
  x: -0.42860301970851935
  y: -0.158937822197668
  z: -0.2751503382204404
N 2
  group: actors
  x: 0.4188857939821254
  y: 0.13267147280573745
  z: -0.4111918528420553
N 3
  group: actors
  x: 0.7000554362043598
  y: -0.37309715595553566
  z: 0.8215559902959177
N 4
  group: actors
  x: 0.3204115077057708
  y: -0.003734454934361879
```

Figure 5.23: GUIBox/JTextArea: the GraphArray created by TensorToGraphArray

Array for the newfrat data, as in Figure 5.23. Since we see only the first few lines, and because of the String format of a GraphArray, there is not much — as far as the contents are concerned — to verify. Just check that it is a GraphArray.
5.9.3.4 Disconnect TensorToGraphArray from ObjectToString

Disconnect TensorToGraphArray from ObjectToString when you are satisfied that the design is producing a GraphArray.

Save your design.

5.10 Visualize the data with Mage

As we did with the two mode data, we can view our one-mode data in the Mage chemical viewer program. To do this, we will need to choose a different output file name. Then we will reconnect the part of the design which converts the GraphArray object into a Mage object, and run the design.

5.10.1 Choose a different Mage output file

First let us choose our output file name, so that we do not over-write our last output file.

5.10.1.1 Change the name of your output file

In the second MtSimpleFileChooser component window, we still have the output file name set to ses_mental.pdb.txt, as seen in Figure 4.57. Let us change
our output file name to reflect the input data we are using. Since the creator of Mage calls his files “kinemages”, let us name our output file newfrat_kin.txt, and save it to the same directory where our input file, newfrat_d1.txt is, as seen in Figure 5.24.

5.10.2 Convert the GraphArray to a Mage

Now let us actually create the Mage image.

5.10.2.1 Connect TensorToGraphArray to GraphArrayToMage

Reconnect (GraphArray)TensorToGraphArray.getOutputGraphArray to GraphArrayToMage.setInputGraphArray(GraphArray)

5.10.2.2 Verify Mage output

Since the GraphArrayToMage bean is still connected to the JTextArea and StringToFile beans, its output should go straight to these beans when we click the Calculate button. Click Calculate now. In the JTextArea component, you should see something like Figure 5.25.

5.10.3 Explore data with Mage

5.10.3.1 Open up your file

If you do not have Mage running, start it. In Mage, open up the newfrat_kin.txt file, as you did in Section 4.5.3.2 for the 2-mode data.
5.10.3.2 Explore the data

Now you should see your data appear as in Figure 5.26.

The first thing you should notice is that unlike the 2-mode data, there are lines connecting the spheres.

Secondly, notice that all the spheres have the same color: red.

Looking to the right of the Mage Graphics window, notice that there are 15 sections — one for each of the weeks, or matrices, in the original newfrat data.

5.10.4 Remove and restore ties

If you click on the ties box under the NEWC0 section, you can make the ties, or lines, go away, as seen in Figure 5.27. Re-set ties to on when you are done.
5.10.5 Animate the data

Now let us move through the 15 weeks, as if they were a motion picture. At the bottom of your Mage window, you should see an Animate checkbox (Figure 5.28). Click on that, and you notice that the view changes to week 2, labelled NEWC1. If you keep clicking, you should move through all 15 various weeks, as seen in Figure 5.29. You should also be able to see that after the first few weeks, the network begins to settle down. For instance, the three spheres in the lower right,
Figure 5.27: Mage: newfrat - 1st week with ties

Figure 5.28: Mage: animating the newfrat data
Figure 5.29: Mage: newfrat - all 15 weeks
numbered 10, 15, and 16, begin to drift off to the right and maintain the same relative positions to each other from that point on, from as early as the third week. Also, spheres 1, 6, 8, and 13 are off to the the upper left from week 1. 2, 3, 7, 11, and 12 move to the upper right. And so on.

But can we tell anything else from the pictures? Looking at the pattern of ties might be helpful, but there are so many of them.

5.10.6 Choose a cut-off level for showing ties

This graph is rather complicated. In fact, it is complete: all balls have connections to all others. This makes sense, since the data are ranked.

Suppose we would like to look only at the closest ties – those people who rank each other the highest. How may we accomplish this with Studio and the socnet beans?

Back on Studio’s DesignBox, open up the Properties dialog for the GraphArrayToMage bean. Set the EdgeWeightCutoff property to something greater than zero. Remember, since we reversed and symmetrized by sum, the largest value of any cell will be 32, which is on the diagonal. So let us try 28, as seen in Figure 5.30. This means that we will draw a line connecting two balls only if its value was 28 or higher. Be sure to press Enter after you have replaced the 0.0 with a 28.0.

Do not click Done yet.

5.10.7 Make all scenes have same color spheres

While we are at it, let us also make another change so that it’s easier for the eye to pick up changes. You may have noticed that all the balls have the same
color, but their color changes from frame to frame. This is the default behavior when using the colors which were included in the GraphArray which TensorToGraphArray produced. But we can have the GraphArrayToMage bean assign different colors, if we like. There are, as of this writing, three other choices, besides GRAPHARRAY COLORS, and they all refer to methods the GraphArrayToMage bean may employ to color the balls:

- **BY FRAME** - This uses the same method as the TensorToGraphArray
uses, although when you have the `GraphArrayToMage` bean do it, you can choose the color of the balls in the first frame by setting the `starting-BallColor` to whatever you like. There are only 20 legal Mage colors, so if there are more than 21 frames the assignment of colors will start over at the 21st frame, so that the 21st frame’s balls will get the same color as the 1st frame, and so on.

- **BY BALL** - Each ball gets a different color (although from frame to frame the same ball gets the same color). This makes it easier to pick out individual movement. Because there are only 20 colors in Mage, if there are more than 20 balls, the 21st ball will get the same color as the 1st ball, and so on.

- **INVARIANT** - The ball colors never change. All balls in the same frame have the same color, and all frames have the same color balls. The color used is the value set in `startingBallColor`.

Let us have the `GraphArrayToMage` bean set the colors, and let us keep them all the same. Set the `BALLCOLORINGMETHOD` property so that its value is **INVARIANT**, as seen in Figure 5.31.

Click **Done** and then click **Calculate** again. In Mage, re-open the `newfrat-kin.txt` file. Click through the frames with the **Animate** button, and notice how the most of the ties have been removed, except a few (Figure 5.32). These are the people who rated each other the closest. Also, notice how these ties are unstable in the first few weeks and then settle down to agreement later in the semester.

Feel free to get more practice by trying other values for `edgeWeightCutoff`, and by using the **BY BALL** value for the `ballColoringMethod`.
5.11 Visualize the data with ChemSymphony

In this section, we will view the Newcomb fraternity data as a PDB molecule using the ChemSymphony beans we’ve already placed in our design in Section 4.6.

Once again, this is more a demonstration of the possibilities of building on each others’ work by using beans, rather than a useful tool. This is because of the limitations both of this particular renderer, and of the PDB format itself.
Figure 5.32: Mage: newfrat - all 15 weeks (\texttt{T2GA.edgeWeightCutoff = 28.0}, \texttt{T2GA.ballColoringMethod = BY FRAME})
5.11.1 Switch to PDB format

To switch to viewing the PDB data, we will need to re-connect our data flow from the Mage part of the flow design to the PDB part.

5.11.1.1 Disconnect TensorToGraphArray and GraphArrayToMage

Disconnect GraphArrayToMage from TensorToGraphArray.

5.11.1.2 Connect TensorToGraphArray to GraphArrayToPDB

Connect (GraphArray)TensorToGraphArray.getOutputGraphArray() to GraphArrayToPDB.setInputGraphArray(GraphArray).

Now, instead of converting a GraphArray to a kinemage file, we will be converting it to a PDB file.

5.11.2 Direct PDB output to JTextArea window and to a file

As we did with GraphArrayToMage, we will verify the output of GraphArrayToPDB, by viewing it in our JTextArea window. This bean is already connected.

We would also like to save our PDB output to a file, via the StringToFile bean. This, too, is already connected. All we need to do is change the output file name.

5.11.2.1 Change the name of your output file

Now, let us change the name of our output file, since it is going to contain PDB data, not a kinemage. In your second MtSimpleFileChooser, choose a new file.
Instead of `newfrat_kin.txt`, change your file name to `newfrat_pdb.txt` (Figure 5.33). Now your TensorToGraphArray connection in your DesignBox should look like Figure 5.34.

![Diagram showing connection between T2GA and GA2PDB](image)

Figure 5.34: DesignBox: TensorToGraphArray (T2GA) reconnected to GraphArrayToPDB (GA2PDB)

Save your design.

### 5.11.2.2 Re-assert the Chemical Data Format

Now, just as we did in Section 4.6.3.5, let us make sure that ChemSymphony’s RenderBasic bean knows that it will be receiving PDB-formatted data: Place
Figure 5.35: GUIBox: Assuring the RenderBasic ChemicalDataFormat is set to PDB by hitting Enter again in the JTextField your cursor in the JTextField in the GUIBox, and hit Enter (Figure 5.35). This will call call RenderBasic.setChemicalDataFormat("PDB") again.

5.11.2.3 Double-check your RenderControl’s settings

If you closed and re-loaded your design since working on the 2-mode tutorial (as I did), you will notice that the ChemSymphony beans have lost their settings. This is because we are using an older version of the beans which do not conform to the newer JavaBean long-term persistence specification. Thus, your RenderControl’s settings will be set back so that DisplayStyle is WireFrame, AtomTypes is Empty, and Atomic Radius Factor is at 50%, and so on. We can get around this by simply resetting these to their preferred values as we did in Section 4.6.4.2. Go ahead and do that now.

5.11.3 Test your design

Now let us view our data in PDB format.

Click on the Calculate button, and see the output in your JTextArea and RenderBasic windows, as in Figure 5.36. A close-up of the JTextArea only
Figure 5.36: GUIBox: PDB output of Newcomb fraternity data

may be seen in Figure 5.37 and a close-up of the ChemSymphony RenderBasic window (with black background removed) in Figure 5.38.

5.11.4 Change viewed frame

Notice that with the PDB data format, we can only see one frame at a time. However, with the socnet bean, TensorToPDB, you can choose which frame you wish to view.

5.11.4.1 Change GraphArrayToPDB’s GraphToExtract property

Right-click on the GraphArrayToPDB bean’s icon in the DesignBox, and choose Property. In the dialog box that pops up, make sure that the Proper-
Figure 5.37: GUIBox/JTextArea: PDB output of Newcomb fraternity data

ties tab is showing.

In the **GraphArrayToPDB** bean, the frames are 0-indexed. This means that in the case of the Newcomb fraternity data, the frames (graphs) are numbered 0 to 14, instead of 1 to 15. Currently the value of the **GraphToExtract** property is 0, meaning the first graph. Change this value to 14, as in Figure 5.39. This is your way of telling the bean that instead of viewing the first graph, you would like to view the last.

Remember to hit the **Enter** key after typing in the 14. Unless you hit **Enter**, the data will not actually register in the properties.

Click **Done**, and run the design again. As you do this, you will notice the effect of the **GraphArrayToPDB** rotating through using the four atoms, O, N, H, and C. Thus, on this 2nd click of the **Calculate** button, your atoms now appear...
Figure 5.38: GUIBox/RenderBasic: PDB output of Newcomb fraternity data as blue Nitrogen (N) atoms. Click Calculate three more times, until the bean rotates around to producing red Oxygen (O) atoms, again. Your RenderBasic window should now reflect the data from the last (15th) of the matrices in the Newcomb data, as in Figure 5.40 (although I’ve removed the black background of the RenderBasic window).

Save your design.

5.11.5 Compare Mage and PDB images

If you like, you may compare the PDB image you just created of the last week (NEWC15) of the Newcomb fraternity with the Mage image of the same data, with ties removed (and again, I removed the black background). These two images may be seen here in Figures 5.40 and 5.41.
5.12 Conclusion

This concludes the tutorial for creating a 1-mode data flow design with Studio, the socnet beans, Mage, and ChemSymphony.

If you would like, you may expand on this design by experimenting with adding another MtSimpleFileChooser and/or JTextArea so that both Mage and PDB formats may be viewed/created with the same click of the Calculate
Figure 5.40: GUIBox/{\texttt{RenderBasic}}: Viewing matrix NEWC15 (16th week) of Newcomb fraternity data

Alternatively, import and view your own 1- or 2-mode datasets, now that you have created two useful tools for quickly creating images from them.
Figure 5.41: Mage: Viewing matrix NEWC15 of Newcomb fraternity data
CHAPTER 6

User Reference

6.1 Introduction

This chapter serves as a reference for users of the beans created for this project.

Section 6.2 lists the fifteen beans programmed for this dissertation project. Input methods are specified for each bean, along with callback methods and properties. The output event is not included because it is the same for every Socnet bean: PropertyChange.PropertyChange. For input and callback methods, required input parameters and outputs returned are specified in parentheses similar to Java method syntax (c.f. Section 3.3.2). Defaults for properties are specified. The basic properties inherited from StackLogger and FunctionBean are included.

6.1.1 Note on Terminology: Properties vs Accessor methods

As explained in Section 3.4.1 beans have properties. Some properties are considered input properties, and these are things we give to the bean so that it may begin calculating. Other properties are considered output properties, and the methods we use to retrieve these are called callbacks.

In explaining how to interact with a bean to give it or retrieve from it some data, I may use the property lingo (e.g., ”set the INPUTTENSOR property with
the **Tensor** you got from the previous bean”). Alternatively, I may refer to the setter or getter accessor methods which access this property (e.g., ”call **setInputTensor(Tensor)**, using the **Tensor** you got from the previous bean”). These are equivalent statements.

### 6.1.2 Review of basic data processing cycle

All Socnet beans descend from the Java class, **FunctionBean**, and have its data processing cycle. All **FunctionBeans** have this basic cycle: First, input properties are set with Java input method (procedure) calls. Second, when all input properties have been set, the bean either begins calculating automatically, or it is manually set running. Third, when the bean is done calculating, it sends out notification to any other beans which are waiting for its data. This notification is called the **PropertyChange.PropertyChange** event, or simply the *output event*. When the downstream beans receive this message, they make a *callback* to get the output data (the output property) from the upstream bean.

### 6.1.3 Explanation of Tables

#### 6.1.3.1 Input Property Tables

The ”Input Properties” tables below each have 5 columns: property, input data type, req’d, default, and Studio conn.. The rows in these tables are the properties (which may, as mentioned above, be accessed by their corresponding ”set” methods by the same names). The property therefore contains the property name. The *input data type* column contains the class or primitive data type which must be sent using the property’s setter. Therefore, if you see a property, **NewMatrixLabels** and its input data type is **StringList**, then the setter for this
input property is `setNewMatrixLabels(StringList)`. The third column, *req’d*, shows whether the bean requires this property to be set in order for calculation (whether manual or nautomatic) to begin. The fourth column, *default*, shows what the value for that property will be before the user sets it. Some defaults are fine as they are, and therefore do not require any more intervention from the user, unless the user wants to set the value of the property to something else. Other defaults (such as null) will *not* do, and the property must be supplied with a value or object. Finally, the fifth column, *Studio conn.*, refers to whether I have configured the Studio (via the *socnet.xml* palette — c.f. Section 2.4.3) so that the bean shows a connector for this input property, when using Studio. However, having configured Studio so that a connector shows automatically for an input property does not necessarily mean that the property is required for calculation (see Section 6.2.2 on *ExpandTensor* for an example of this). Nor does *not* having a connector mean that a connector cannot be added, in cases where you do want to use an input property that is not normally used. (Section 2.5.6 shows the set up of connectors for the third-party ChemSymphony beans on the Studio palette, and Section 4.2.3.6 covers the setup the Socnet beans.)

Properties which do not appear as red or blue connectors on Studio bean icons will still be visible from within the properties dialog window for each bean. For instance, *TensorToGraphArray*’s `ColVertexLabel` property does not normally have an input connector showing (c.f. Table 6.31). However, this property will still show up in Studio’s property dialog for this bean (c.f. Figure 4.47). Furthermore, this property may be given a connector in Studio (as shown in Sections 2.5.6 and Section 4.2.3.6) should the user for some reason want to change the row labels for consecutive data sets, perhaps as might be the case if a number of data sets are being run through sequentially.
6.1.3.2 Callback Property Tables

Each bean also has a table of callback properties. This table has three columns: 
property, output data type, and Studio callback. The first column lists the callback property name. The second lists the data type (whether a Java object class or a primitive data type such as an integer or boolean) which you will get back from the callback. The third column is analogous to the input table’s Studio conn. column. However, in Studio, callback methods are not shown as ”output connectors” on the icons for the Java beans. Instead, there is only one connector used for output, which corresponds to the signal (event) the bean sends out when it is done calculating. This signal is PropertyChange.propertyChange(). Then the ”down-stream” beans make a call back to the bean for data they want. The getter methods (callbacks) are instead shown in a dialog box at the time of connecting the beans together.

An example of this may be seen in the use of the Adapter Wizard in Figure 4.9, where the user is connecting up a non-Socnet Java bean, the MtSimpleFileChooser, to the ReadDL bean. In the window the reader may see MtSimpleFileChooser’s callbacks, getFile() and getAbsolutePath().

Another example may be seen in Figure 4.35, where the five normal callbacks for the Socnet bean, SVD, are listed.

There are usually more callbacks available than are listed in Studio’s Adapter Wizard. But Studio’s palette has been configured so that only the relevant getters will be listed as callbacks for this bean. This is a convenience for the user.

For instance, getInputTensor would not normally be used as a callback. However, if you wanted this to appear in the list of callbacks shown in the Adapter Wizard, you could do so, again, by going to the Callbacks tab of the Properties
dialog window for this bean, and putting a check next to that method.

6.1.3.3 The most important properties

Finally, in each input properties and callback properties table, properties will be normally divided by a single line. The properties above this line are the properties that a user of the bean in Studio, or a programmatic user, will normally be concerned with setting in order to get the bean to calculate, or in order to get data out of the bean. For instance, in the properties for CollapseTensor, the reader may see that while all properties listed are required for calculation, nonetheless most of them have defaults already set for the user. Only InputTensor is not set. Therefore, InputTensor is the only property for this bean that a user would normally be worried about setting. Indeed, intuitively speaking, this is all the CollapseTensor should normally need: the Tensor to collapse.

When there is no line separating important properties from lesser ones (such as is the case in the callback properties for StringToFile, Section 6.2.12), this means that none of them are normally used. This will also be noted in the text.

6.1.4 LogLevel property

Concerning the LogLevel property of all Socnet beans: All Socnet beans are FunctionBeans, and all FunctionBeans are StackLoggers. Therefore, all Socnet beans have the LogLevel property.

Setting this property to other than OFF for a bean will result in text messages coming up in the Console window of Studio. Usually, in debugging (usually used only by the programmer of a new bean), levels of FINER or FINEST may be used. For users of the beans in Studio, WARNING and SEVERE are sufficient
<table>
<thead>
<tr>
<th>Level</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEVERE (highest)</td>
<td>If you only want to be notified of errors which stop the bean’s calculation</td>
</tr>
<tr>
<td>WARNING</td>
<td>For warnings which are serious, but will not stop calculation</td>
</tr>
<tr>
<td>INFO</td>
<td>Basic informational messaging about the calculation going on in the bean</td>
</tr>
<tr>
<td>CONFIG</td>
<td></td>
</tr>
<tr>
<td>FINE</td>
<td></td>
</tr>
<tr>
<td>FINER</td>
<td></td>
</tr>
<tr>
<td>FINEST (lowest)</td>
<td>If you want to see almost everything that the beans are doing</td>
</tr>
<tr>
<td>OFF</td>
<td>This is the default. Even warnings will not be displayed</td>
</tr>
<tr>
<td>ALL</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Values for the LogLevel Property

to give feedback when there are problems.

Settings are inclusive, from lowest to highest. The "level" refers to the level of granularity that you want. SEVERE is the coarsest level, and includes only SEVERE messages. When LOGLEVEL is set to FINEST, all messages of level FINER to SEVERE will also be displayed. Setting a level of CONFIG includes CONFIG, INFO, WARNING, and SEVERE, but excludes FINE, FINER and FINEST messages from showing, and so on. Therefore, setting a bean’s LOGLEVEL property to FINEST will mean that all seven levels are included.\(^1\)

\(^1\)It is possible for a programmer to add other levels, which is why there is also a level of ALL. However, in most normal usage, ALL and FINEST would be equivalent.
It should also be noted that the bean’s programmer has to manually insert messages at given log levels into the code for the bean in order for this to work. The StackLogger class programmed for this project (and from which all FunctionBeans – and therefore all Socnet beans– descend) makes this somewhat easier for the programmer. This is intended as an inducement to the programmer to use the logging facility of StackLogger, in the end to help the end user. But the user should be aware that it is still up to the programmer to manually insert these messages into his/her code. Therefore the usefulness of this feature depends on the programmer of the bean.

6.1.5 AutoCalculate and AutoClearInputs properties

AutoCalculate and AutoClearInputs are properties all Socnet beans inherit from FunctionBean.

When AutoCalculate is set to TRUE – as it is for most beans – this means that the bean will begin calculating as soon as all required inputs have been set. Beans which do not normally have AutoCalculate set to TRUE are beans which are normally put at the beginning of a data flow pipeline, such as ReadDL: Just because the bean’s INPUTDFile property is set does not necessarily mean we want the data flow pipeline to begin execution. In the tutorials, for instance, ReadDL’s doCalculate() method is showing (has a connector in Studio), even though this is not a property. doCalculate() is a method that all Socnet beans have (which they inherit from FunctionBean). However, it is normally called automatically by the bean itself, once all properties are set. In a manually-run bean like ReadDL, AutoCalculate is set to FALSE, and therefore, the bean has been configured in Studio so that its doCalculate() method is showing, and may be connected to a button or some other signalling device in order to get the
bean running manually. (c.f. Section 4.2.2).

AutoClearInputs tells the bean whether or not to clear all of its inputs once it has finished calculating. This helps the bean to get ready for another processing run. It is especially helpful for beans which have more than one input property to be set each time. If a bean had 2 or more input properties to be set, and did not clear them at the end of a processing cycle, then as soon as only one of these were set a second time, the bean would begin calculating again without waiting for the 2nd (or more) properties to also be set. This is because the inputs from the previous cycle would still remain. Therefore, for most beans, this property is set to TRUE, which tells the bean to clear out all input properties in the final stages of its data processing cycle.

However, there are cases where AutoClearInputs is set to FALSE, by default. This usually is for starting and terminating beans. ReadDL has AutoClearInputs set to FALSE because after running, we do not want to lose the value of its InputDLFileName property. If we did, we would have to re-enter the file name each time we ran the bean. Similarly, StringToFile has AutoClearInputs also set to FALSE for the same reason: we want to keep the value of its OutputFileName property. (Nevertheless, its AutoCalculate property is still set to TRUE.)

6.1.6 Nomenclature

Concerning the naming of several of the beans: There may be some confusion over the naming of the CollapseTensor, ExpandTensor, and StackTensors beans.

CollapseTensor and ExpandTensor respectively collapse and expand the matrices of a Tensor. StackTensor stacks two tensors on top of each other to
become one Tensor. If CollapseTensor and ExpandTensor are working with matrices, and StackTensor with a Tensors, then why are not CollapseTensor and ExpandTensor called "CollapseMatrices" and "ExpandMatrices"?

The reason for this has to do with planning for future beans. Tensors consist of matrices. While up to this point the architecture of Tensors has not been discussed in depth, the matrices a Tensor contains are actually objects of the class socnet.SMatrix. Future beans may pass between them only an SMatrix, not a whole Tensor. Though I have used Tensors for passing data between all my beans, other authors may not feel so inclined.

Therefore, the naming of the bean follows the inputs given to it. Even though CollapseTensor and ExpandTensor are collapsing SMatrixes within a Tensor, what these beans are given as inputs is the Tensor. Therefore, the point of view is of the Tensor. There is an input Tensor and an output Tensor, and the output Tensor is collapsed with respect to the input Tensor.

In this way, if an author of another bean uses only SMatrixes as inputs, the name of his or her bean could have “SMatrix” in the name.

6.2 Bean Reference

Following are the input properties, callback properties, and general guidelines for using each of the 15 Socnet beans.

6.2.1 CollapseTensor

Collapses the matrices of a Tensor into a single matrix.

Optionally, you may first call setMethod(String), in order to tell CollapseTensor how to collapse the matrices. If you use "VERTICAL", the matrices will
<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTTENSOR</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>METHOD</td>
<td>String</td>
<td>Y</td>
<td>&quot;VERTICAL&quot;</td>
<td>N</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCLEARINPUTS</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.2: Input Properties for CollapseTensor

be collapsed vertically, so that the matrices are made to be additional rows in the first matrix. (Matrix 0 on top, then Matrix 1, then Matrix 2 below, etc.) If you use "HORIZONTAL," the matrices will be collapsed horizontally, making for additional columns. If you do not call setMethod(), the default is "VERTICAL."

Now set the input **Tensor** you want to collapse the matrices of. Normally, AUTOCLEARINPUTS is set, so the bean will automatically start running. Wait for calculate to return, or wait for the PropertyChange.PropertyChange event.

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTTENSOR</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>INPUTTENSORMATRIXLABELS</td>
<td>StringList</td>
<td>Y</td>
</tr>
<tr>
<td>METHOD</td>
<td>String</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.3: Callback Properties for CollapseTensor

After calculation is finished, you are ready to get your output. Call getOutputTensor() to get the new **Tensor** which contains the original matrices collapsed
into a single matrix.

Optionally, you may also call `getInputTensorMatrixLabels()` to get the original names of the separate matrices from the original input `Tensor`. This is useful if you want to send the `INPUTTensor`’s original matrix names to `ExpandTensor`, so `ExpandTensor` may label the re-expanded matrices using the original matrix names.

An example of the use of `CollapseTensor` may be seen in Section 5.6.1

### 6.2.2 ExpandTensor

Expands the first matrix of a `Tensor` into several matrices in a new `Tensor`

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>INPUTTensor</code></td>
<td><code>Tensor</code></td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td><code>NEWMatrixLabels</code></td>
<td><code>StringList</code></td>
<td>Y/N†</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td><code>NumberOfNEWMatrices</code></td>
<td><code>int &gt; 0</code></td>
<td>Y/N‡</td>
<td>0 (zero)</td>
<td>N</td>
</tr>
<tr>
<td><code>RequireMatrixLabels</code></td>
<td><code>boolean</code></td>
<td>Y</td>
<td>FALSE</td>
<td>N</td>
</tr>
<tr>
<td><code>Method</code></td>
<td><code>String</code></td>
<td>Y</td>
<td>&quot;VERTICAL&quot;</td>
<td>N</td>
</tr>
<tr>
<td><code>LogLevel</code></td>
<td><code>Level</code></td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td><code>AutoCalculate</code></td>
<td><code>boolean</code></td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td><code>AutoClearInputs</code></td>
<td><code>boolean</code></td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

†if `RequireMatrixLabels = TRUE`, then Y, else N
‡if `RequireMatrixLabels = TRUE`, then N, else Y

Table 6.4: Input Properties for `ExpandTensor`

Optionally, you may first call `setMethod(String)`, in order to tell `ExpandTen-
how to collapse the matrices. If you use "VERTICAL", the matrices will be expanded vertically, so that the matrices are taken from the rows in the first matrix. (Matrix 0 on top, then Matrix 1, then Matrix 2 below, etc.) If you use "HORIZONTAL," the matrices will be created from the columns of the original matrix. If you do not call setMethod(), the default is "VERTICAL."

If you set RequireMatrixLabels to TRUE, then you must set NewMatrixLabels. In this case NumberOfNewMatrices will be ignored if you set it beforehand, since the bean will calculate it implicitly by counting the number of matrices in the NewMatrixLabels property.

On the other hand, if you leave RequireMatrixLabels set to FALSE, then you must set NumberOfNewMatrices (and NewMatrixLabels will be ignored, even if you set it as well). NumberOfNewMatrices must be 1 or more.

Now set the input Tensor you want to expand. Normally, AutoCalculate is set, so the bean will automatically start running.

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputTensor</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>Method</td>
<td>String</td>
<td>N</td>
</tr>
<tr>
<td>RequireMatrixLabels</td>
<td>boolean</td>
<td>N</td>
</tr>
<tr>
<td>NumberOfNewMatrices</td>
<td>StringList</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.5: Callback Properties for ExpandTensor

To get your output, call getOutputTensor() to get the new Tensor which contains the original matrix expanded out into the number of matrices you chose.
Each matrix will have a label assigned to it, created automatically as `mat0`, `mat1`, etc., or will be assigned one of the labels you assigned with the `NewMatrixLabels` property.

Optionally, you may also call `getInputTensorMatrixLabels()` to get the original names of the separate matrices from the original input `Tensor`. This is useful if you want to re-expand the matrices of the `Tensor` with `ExpandTensor`, and want to preserve the matrix names.

Section 5.8.1 demonstrates the use of `ExpandTensor`.

### 6.2.3 GraphArrayToMage

`GraphArrayToMage` converts a `GraphArray` object to a *kinemage* – a textual description of a chemical molecule in a format suitable for viewing with the program, Mage (see Section 2.6). The kinemage output is given as a `String` object.

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>INPUTGraphArray</code></td>
<td><code>GraphArray</code></td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td><code>BALLColoringMethod</code></td>
<td><code>String</code></td>
<td>Y</td>
<td>&quot;INARIANT&quot;</td>
<td>Y</td>
</tr>
<tr>
<td><code>EdgeWeightCutoff</code></td>
<td><code>double</code></td>
<td>Y</td>
<td>0.0</td>
<td>Y</td>
</tr>
<tr>
<td><code>StartingBallColor</code></td>
<td><code>String</code></td>
<td>Y</td>
<td>&quot;red&quot;</td>
<td>N</td>
</tr>
<tr>
<td><code>LOGLEVEL</code></td>
<td><code>Level</code></td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td><code>AutoCalculate</code></td>
<td><code>boolean</code></td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td><code>AutoClearInputs</code></td>
<td><code>boolean</code></td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.6: Input Properties for `GraphArrayToMage` (GA2M)
Table 6.7: Callback Properties for **GraphArrayToMage** (GA2M)

To use the bean, you first optionally set its **BALLCOLORINGMETHOD**, **EDGEWEIGHTCUTOFF**, and **STARTINGBALLCOLOR** properties, if you do not like any of the defaults available for these. Then you set your **INPUTGRAPHARRAY** and calculation will begin automatically. When calculation has finished, make a call back to retrieve the **OUTPUTKINEMAGE** (as a **String**).

The first of the three options, **BALLCOLORINGMETHOD**, may take one of four values. The default is "**INVARIANT**," meaning that all balls in all graphs in the output kinemage will have the same color. The second value available is "**BY FRAME**," meaning all the balls in each frame of the output kinemage will be assigned all the same color, and each frame’s balls will have a different color, rotating through the colors the bean has available to it. The third available value is "**BY BALL**," meaning that the colors will be assigned by ball, so that as each ball is drawn, it will be assigned the next available color. If the colors run out, then the first color will be used again, and so on. The last available value is "**GRAPHARRAY COLORS**," which means that the balls will be assigned whatever colors are already assigned to them in the input **GraphArray**.

The bean contains a set of colors which are the full set of legal colors available for a kinemage, according to the kinemage format specification (Richardson 229).
These ball colors are: red, blue, green, cyan, yellow, magenta, white, pink, orange, purple, sky, brown, gray, gold, yellowtint, sea, pinktint, bluetint, greentint, and hotpink. These colors are listed in the order that the bean assigns them to the balls. Thus, the color of the first ball will always be red unless BallColoringMethod or StartingBallColor are altered (see below) from their defaults.

If BallColoringMethod is "BY BALL" then the second ball in the first frame will be blue, the third ball green, and so on. If there are more balls then colors, then the coloring will rotate around to red again. Moving on to the next frame, the first ball in the second frame will take the next ball color after the color assigned to the last ball in the previous frame.

If BallColoringMethod is "BY FRAME" then all balls in the same frame will also be red, and the first (and subsequent balls) in the second frame will be blue, all the balls in the third frame green, and so on.

If BallColoringMethod is "INVARIANT," then all the balls in all the frames will be red. If BallColoringMethod is "GRAPHARRAY COLORS," then the colors are taken from the input GraphArray.

The second of the three options, EdgeWeightCutoff is a type of filter. It refers to the point at which a connector/bond will be entered in to the kinemage. The default is 0.0, which means that if the edge weight connecting vertices a and b is greater than 0.0, then an edge (or bond) will be created in the output kinemage format. Any 0 or negative values will result in no bond being drawn.

This feature is useful in weighted data sets where the researcher wishes to see only the strongest weights (bonds) in a complex network. In a binary structure, all "weights" are either 0 or 1, so with an EdgeWeightCutoff of 0.0, all connections will automatically be included.
The last of the three options is `STARTINGBALLCOLOR`, which may take any of the ball color values listed above.

`GraphArrayToMage` is put to use in the tutorials in Sections 4.5.1.1 and 5.10.2.2.

### 6.2.4 GraphArrayToPDB

This bean converts a `GraphArray` object to PDB (Protein Data Bank Atomic Coordinate Entry) format ([Protein Data Bank](https://www.rcsb.org/pdb/), 1996). The PDB output is given as a `String` object.

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InputGraphArray</code></td>
<td><code>GraphArray</code></td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td><code>EdgeWeightCutoff</code></td>
<td>double</td>
<td>Y</td>
<td>0.0</td>
<td>Y</td>
</tr>
<tr>
<td><code>GraphToExtract</code></td>
<td>int</td>
<td>Y</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td><code>LogLevel</code></td>
<td><code>Level</code></td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td><code>AutoCalculate</code></td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td><code>AutoClearInputs</code></td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.8: Input Properties for `GraphArrayToPDB` (GA2PDB)

To use `GraphArrayToPDB`, set your `GraphToExtract` to something greater than 0 (the first graph), if the input `GraphArray` contains more than one graph.

Next, set your `InputGraphArray`, and calculation will commence. When calculation has finished, call back to get the `OutputPDB` as a `String` object.

This bean does not have all the coloring facilities available to it as `GraphAr-
<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTPDB</td>
<td>String</td>
<td>Y</td>
</tr>
<tr>
<td>EdgeWeightCutoff</td>
<td>double</td>
<td>N</td>
</tr>
<tr>
<td>GraphToExtract</td>
<td>int</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.9: Callback Properties for GraphArrayToPDB (GA2PDB)

rayToMage, because the PDB format does not allow for the style (color) data such as which atoms are assigned which colors. Instead, the bean assigns atom types – O, H, N, and C – to the vertices from the input GraphArray. While the bean does not currently support the ability to specify other atom types (and beginning atom type) in the way that GraphArrayToMage specifies colors, this facility could be added to the bean in the future.

The reader may find demonstrations of the use of GraphArrayToPDB in Sections 4.6.1 and 5.11.1.2 of the tutorials.

6.2.5 ObjectToString

This bean takes any input object, and uses that object’s toString() method to produce a String representation of that object. It is highly useful when combined with the StringToFile bean to save the output of any bean to a text file.

First set the InputObject with setInputObject(Object). Calculation will start automatically.

When calculation has finished, call back to getOutputString() to get the output OutputString property.

ObjectToString is introduced in Section 4.2.3 of the 2-mode tutorial, and
### Table 6.10: Input Properties for **ObjectToString**

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTOBJECT</td>
<td>Object</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>LogLevel</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AutoCalculate</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AutoClearInputs</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

### Table 6.11: Callback Properties for **ObjectToString**

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputString</td>
<td>String</td>
<td>Y</td>
</tr>
</tbody>
</table>

is used throughout both tutorials.

#### 6.2.6 OptScale

The **OptScale** function bean rescales the row and column vectors from a singular value decomposition. This function is the first post-decomposition transformation used in correspondence analysis, and is outlined in (Weller and Romney, 1990, pp. 23,59-62). This transformation scales the scores according to the original data so that the row and column scores can be compared meaningfully.

To use the bean, set the INPUTO property, using a copy of the original data matrix (before normalization and SVD). Then set the input row and column scores from the SVD (the OUTPUTU and OUTPUTV properties from the SVD bean), by setting the INPUTU and INPUTV properties. Once all three of these

233
are set, the bean will begin calculating.

When calculation has finished, call back to get the OutputX and OutputY properties, which represent the optimally-scaled row and column scores, respectively. These will be represented/output as Tensors. OptScale is usually followed by ReWeight as part of a correspondence analysis procedure.

Examples of the use of OptScale may be seen in the tutorials in Sections 4.3.3.1 and 5.7.2.2.
6.2.7 ReadDL

This bean reads in a Data Language-formatted text file (Borgatti, 1985) and outputs either a Tensor or DL object.

<table>
<thead>
<tr>
<th>property</th>
<th>input</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTDLFileName</td>
<td>String</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>LogLevel</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AutoCalculate</td>
<td>boolean</td>
<td>Y</td>
<td>FALSE</td>
<td>N</td>
</tr>
<tr>
<td>AutoClearInputs</td>
<td>boolean</td>
<td>Y</td>
<td>FALSE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.14: Input Properties for ReadDL

Set the Data Language-formatted input file name you want to read, with setInputDLFileName. Then manually start this bean running with a call to doCalculate. When calculation has finished, call getOutputDL() to get the DL object or to getOutputTensor() to get the output Tensor which contains the data the original input file represented. A Tensor will print out (via its Tensor.toString() method) differently than a DL will: If you want Data Language-formatted output, use getOutputDL. Since a DL is a Java object which descends from Tensor that means that otherwise a DL will function as a Tensor, so that either may be used to pass on to down-stream beans: it will make no difference to them. However, the tutorial chapters use getOutputTensor() for simplicity.

Though AutoCalculate is on for most Socnet beans, it is off for ReadDL because we do not want to calculate automatically as soon as the input file name is set. This bean is intended normally to stand at the beginning of a design (i.e.,

---

2A DL is a Tensor, but outputs its contents differently.


<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputTensor</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>OutputDL</td>
<td>DL</td>
<td>N</td>
</tr>
<tr>
<td>InputDLFileName</td>
<td>String</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.15: Callback Properties for ReadDL

there are no other Socnet beans before it), and to be run manually via a signal sent to its doCalculate method. For instance, in a graphical environment, this can be done with a JButton.

However, in a design where ReadDL is not the first Socnet bean (for instance, where it is being used to sequentially read several or many input files, in a batch style of operation), then AutoCalculate should be set to TRUE.

For most other Socnet beans, this will be TRUE so as to ready the bean for a new data processing cycle. However, for this bean we want it FALSE because there is only one input – the input file name. Automatically clearing the inputs would mean that each time we ran the design, all inputs including the InputDLFileName – the name of the file we are going to be reading the next time the design is run – would be cleared. Thus, we could not run the design a second time without re-entering the name of the input file. Since this would be inconvenient, we leave this set to FALSE for this bean.

ReadDL will parse all DL-formatted (Borgatti, 1985) files except list and array-based formats. It understands the following DL words:

dl, n, nr, nc, nm, format, fullmatrix, diag(onal), present, absent, upperhalf, lowerhalf, blockmatrix, nodelist1, nodelist1b, nodelist2, rank-

236
edlist1, rankedlist2, edgelist1, edgelist2, edgearray1, lab(els), col-
(umn), row, r, v, data, !, to, all, mat(rix), lev(el), embedded, alpha, title,
yes, true, no, and false.

Keywords may be either in upper or lower case. As with normal DL, they
may be separated by a space, “,”, or “=”. ReadDL correctly implements the
use of the ! and alpha keywords, as well.

ReadDL can accept fullmatrix, upperhalf, lowerhalf, and blockmatrix formats.
For these formats, labels (matrix, column, or row) may be either specified in the
header or embedded. Diagonals may be present or absent.

The keywords nodelist1, nodelist1b, nodelist2, rankedlist1, ranked-
list2, edgelist1, edgelist2, and edgearray are understood but these data
formats are not yet supported. Depending upon interest, they may be in a future
version.

Ends-of-line should be in the format of the system that ReadDL is running
on. ReadDL will automatically detect the system native end-of-line format and
expect the file to be in this format. For UNIX/Linux, ends-of-line are delineated
by a single linefeed character (ASCII 10, or 0xa). For Microsoft Windows-based
machines, it is a single carriage return character followed by a linefeed (ASCII
13, ASCII 10 ; 0x0d, 0x0a). Macintosh expects a single carriage return: ASCII
13, or 0x0d.

ReadDL’s use in the tutorials begins in Section 4.2.2. Please see the UCINET
software online documentation (Borgatti et al., 2003) for more information on the
DL language format.
6.2.8 ReverseTensor

ReverseTensor reverses the values in the cells of a Tensor. It does this by taking the sum of the highest value of any cell and the lowest value of any cell, and then reassigning to all cells the value gotten by subtracting the current value from the sum.

Set the input Tensor with setInputTensor(). Calculation will automatically start.

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTTENSOR</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCALCULATE</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCLEARINPUTS</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.16: Input Properties for ReverseTensor

When calculation is finished call back for the reversed Tensor with getOutputTensor.

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTTENSOR</td>
<td>Tensor</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 6.17: Callback Properties for ReverseTensor

ReverseTensor is used in Section 5.4.1 in the 1-mode tutorial.
6.2.9 ReWeight

The ReWeight bean is the second of two post-SVD transformations performed in a correspondence analysis. It takes as its inputs the optimally-scaled row and column scores already output from the first post-SVD transformation, an optimal scaling (such as performed by OptScale, covered in Section 6.2.6). ReWeight reweights the row and column scores from a singular value decomposition as a function of the size of their respective singular values. This procedure is reviewed by Weller and Romney (1990, p.62). The default is to multiply by the square root of the singular value, although there are other methods. (Default-only is supplied at this time). Once the scores are reweighted, they may be viewed in the same space.

<table>
<thead>
<tr>
<th>property</th>
<th>data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTD</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>INPUTX</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>INPUTY</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCALCULATE</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCLEARINPUTS</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.18: Input Properties for ReWeight

To use the bean, send it the eigenvalues from the original SVD by setting the InputD property with the SVD bean’s OutputD output Tensor. Also set the input row and column scores — the InputX and InputY properties. These may be taken from the output Tensors given from OptScale’s OutputX and
<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTXw</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>OUTPUTYw</td>
<td>Tensor</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 6.19: Callback Properties for ReWeight

Once these three properties are set, calculation will begin automatically.

When calculation has finished, call back to get the OUTPUTXw and OUTPUTYw properties, which represent the reweighted row and column scores, respectively. These will be represented/output as Tensors.

Examples of the use of ReWeight may be seen in the tutorials in Sections 4.3.4.1 and 5.7.3.1.

6.2.10 SquareRootNorm

This bean performs a pre-SVD transformation normally done in correspondence analysis to remove the “marginal effect.”

SqRtNorm performs a type of normalization on each of the separate matrices of a Tensor. Each cell is divided by the square root of the product of its respective column-sum and row-sum ([Weller and Romney, 1990](#), pp. 59-60).

To use SqRtNorm, set your input Tensor (INPUTTensor property). Calculation will begin automatically.

When calculation has finished, call back to get OUTPUTTensor, a Tensor containing the normalized matrices.
### Table 6.20: Input Properties for \textbf{SqRtNorm}

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTTensor</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>LOGLevel</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AutoCalculate</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AutoClearInputs</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

### Table 6.21: Callback Properties for \textbf{SqRtNorm}

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTTensor</td>
<td>Tensor</td>
<td>Y</td>
</tr>
</tbody>
</table>

Examples of the use of the \textbf{SqRtNorm} bean may be seen in Sections \ref{sec:4.3.1.1} and \ref{sec:5.7.2.1}.

#### 6.2.11 StackTensors

Stacks two \textbf{Tensors}, combining them into a single \textbf{Tensor} object.

Set the \textbf{Tensor} you want to be on top with \textit{setInputTensor1()}. Set the bottom \textbf{Tensor} with \textit{setInputTensor2}. (The order that these are accessed is irrelevant.) Calculation will automatically start once both are set.

Once calculation has finished, call back to \textit{getOutputTensor()} to get the stacked \textbf{Tensor}.

\textbf{StackTensors} is used in the 2-mode tutorial in Section \ref{sec:4.3.5}
### Table 6.22: Input Properties for StackTensors

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTTENSOR1</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>INPUTTENSOR2</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCALCULATE</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCLEARINPUTS</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

### Table 6.23: Callback Properties for StackTensors

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTTENSOR</td>
<td>Tensor</td>
<td>Y</td>
</tr>
</tbody>
</table>

#### 6.2.12 StringToFile

This bean saves any String to a file.

First set the INPUTSTRING property with a call to `setInputString(String)`. Then set the OUTPUTFILENAME property with a call to `setOutputFileName(String)`. Calculation (saving the String to the file) will begin automatically.

Notice that AUTOCLEARINPUTS is set to FALSE for this bean, since normally we would not want to have to reset the file name each time. However, if you wanted to save to a different file name each time, and in the case where you ”calculated” or provided a different file name for each cycle (i.e., from another bean), you would want to set AUTOCLEARINPUTS to TRUE in that case.

When calculation is finished the file will have been created if necessary, and
Table 6.24: Input Properties for StringToFile

then written to. (If it existed beforehand, the previous contents are added to the file rather than erasing them.)

Since output is to a file, normally this is a terminating bean, and there are no other beans making callbacks. Therefore, I did not set the available callbacks to be shown in Studio. However, they may be, if you like, or have some other purpose for calling back to them after the file has been saved.

Table 6.25: Callback Properties for StringToFile

Ends of line in the output file will be automatically set to the format of the machine the bean is currently running on, i.e., for DOS/Windows, 0x0d 0x0a (13 10); for Linux/Unix, 0x0a (10); for Mac, 0x0d (13).

StringToFile is used throughout Chapters 4 and 5. It is first used in Sec-
### 6.2.13 SVD

The SVD bean performs a singular value decomposition on a Tensor. Normally, a separate SVD is performed on each of the matrices of the Tensor. However, as covered in the 1-mode tutorial, CollapseTensor (Section 6.2.1) and ExpandTensor (Section 6.2.2) may be used to collapse/expand the matrices before/after the SVD so as to perform the SVD on all of the matrices of the Tensor at the same time (in the same space).

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTTensor</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>LOGLEVEL</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCALCULATE</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AUTOCLEARINPUTS</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.26: Input Properties for SVD

Set your input Tensor (INPUTTensor property). Calculation will begin automatically.

When calculation has finished, call back to one or more of the callback methods to get the output properties you desire. OUTPUTU, OUTPUTD, and OUTPUTV contain the U, D, and V matrices — row scores, singular values, and column scores respectively — which one expects as output from a singular value decomposition.

OUTPUTRANKS contains a Tensor with a single matrix, itself containing a
single row. This row contains an array of the ranks of each of the matrices in the original input Tensor.

**OutputDStats** is a Tensor which contains not only the singular values, but also some simple statistics on these singular values, for those interested in knowing how much ”variance” the first $x$ dimensions account for. The rows in this Tensor consist of the dimensions in the output data. The four columns consist of: the singular value itself for this dimension; the percent this singular value represents of the sum of the singular values; the cumulative percent of the singular values up to and including this one; and the ratio of the size of this singular value to the next one.

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTD</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>OUTPUTDStats</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>OUTPUTRanks</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>OUTPUTU</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>OUTPUTV</td>
<td>Tensor</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 6.27: Callback Properties for SVD

Examples of the use of the SVD bean may be seen in Sections 4.3.2.1 and 5.7.2. Correspondence analysis is briefly reviewed in Section 4.3.

### 6.2.14 SymmetrizeTensor

Symmetrizes the values in the cells of each matrix in the Tensor, according to the method chosen by the METHOD property. In symmetrizing, both $x_{i,j}$ and $x_{j,i}$
are assigned the same value. This value is calculated based on the String value of the Method property. The available choices and the corresponding value assigned to both $x_{i,j}$ and $x_{j,i}$ in each case are listed below.

<table>
<thead>
<tr>
<th>string value of Method</th>
<th>float value assigned to both $x_{i,j}$ and $x_{j,i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;MAXIMUM&quot;</td>
<td>$\max(x_{i,j}, x_{j,i})$</td>
</tr>
<tr>
<td>&quot;MINIMUM&quot;</td>
<td>$\min(x_{i,j}, x_{j,i})$</td>
</tr>
<tr>
<td>&quot;AVERAGE&quot;</td>
<td>$\frac{x_{i,j} + x_{j,i}}{2}$</td>
</tr>
<tr>
<td>&quot;SUM&quot;</td>
<td>$x_{i,j} + x_{j,i}$</td>
</tr>
<tr>
<td>&quot;DIFFERENCE&quot;</td>
<td>$</td>
</tr>
<tr>
<td>&quot;PRODUCT&quot;</td>
<td>$x_{i,j}x_{j,i}$</td>
</tr>
<tr>
<td>&quot;DIVISION&quot;</td>
<td>$\frac{x_{i,j}}{x_{j,i}}$</td>
</tr>
<tr>
<td>&quot;LOWERHALF&quot;</td>
<td>$x_{i,j}$ where $i &gt; j$</td>
</tr>
<tr>
<td>&quot;UPPERHALF&quot;</td>
<td>$x_{i,j}$ where $i &lt; j$</td>
</tr>
</tbody>
</table>

Table 6.28: Methods of Symmetrization specified by the Method property

The default value for Method of symmetrization is "MAXIMUM". If you want to change this, set it before setting the InputTensor.

Then, give the bean the Tensor to be symmetrized by setting the InputTensor property with setInputTensor(Tensor). Calculation will begin automatically, unless AutoCalculate is set to FALSE.

Once calculation has finished, call back to getOutputTensor() to get the symmetrized Tensor produced by this bean.

Readers interested in seeing SymmetrizeTensor in action may consult Section 5.5.1 in the 1-mode tutorial.
<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUTTensor</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>Method</td>
<td>String</td>
<td>Y</td>
<td>&quot;MAXIMUM&quot;</td>
<td>N</td>
</tr>
<tr>
<td>LogLevel</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AutoCalculate</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AutoClearInputs</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.29: Input Properties for SymmetrizeTensor

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUTTensor</td>
<td>Tensor</td>
<td>Y</td>
</tr>
<tr>
<td>Method</td>
<td>String</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.30: Callback Properties for SymmetrizeTensor

6.2.15 TensorToGraphArray

This bean converts a Tensor to a GraphArray object. One graph is created in the GraphArray for each matrix in the Tensor. The bean can handle both 2-mode (e.g., actor by event) and 1-mode (e.g., actor by actor) data. The bean will color the vertices differently if it is 2-mode data. Also, if there are any stackpoints in the Tensor (i.e., the Tensor’s matrix is a stack of several matrices which were earlier expanded out), then the rows (or columns) between each stack point will be converted into vertices which have different colors, as well. This is useful for differentiating by color the "row vertices" from "column vertices" in a 2-mode Tensor (c.f. tutorial 1, Chapter 4), or for differentiating the matrices in
a 1-mode, but multi-matrix Tensor which was stacked before a Singular Value Decomposition, and then re-expanded afterwards, with stack points noted (c.f. tutorial 2, Chapter 5).

To use the bean, first decide whether you will be drawing edges between the vertices of your final graph.† In most 2-mode data, you will not, and in most 1-mode data, you will. But there are exceptions in either case. So first check that the INCLUDEWEIGHTS property is set as you desire it. If it is set to TRUE, then the bean will require that setInputEdgeWeights is called: you will have to give the bean edge weights so that it can create edges in the output GraphArray. Therefore, the bean will wait until both the INPUTCOORDINATES and INPUTEDGEWEIGHTS properties have been set before it auto-calculates.

On the other hand, if INCLUDEWEIGHTS is set to FALSE, the bean will require only the INPUTCOORDINATES property be set before it begins calculation.

So set the INCLUDEWEIGHTS property first. The default value is FALSE.

Other optional properties to set are COLVERTEXLABEL, ROWVERTEXLABEL, and EDGELABEL. These are the labels for the column-set and row-set, and edges, respectively. Depending upon which eventual output renderer you use, they may or may not appear. Mage, for instance, will show them as the names of the groups, in the right-hand-side of the Mage window (c.f. Figures 4.55 and 5.27) whereas ChemSymphony will not. These string labels may be set to whatever you prefer. But they must be set before calculation begins.

Then, set the INPUTCOORDINATES with a call to setInputCoordinates(Tensor), giving it your input Tensor.

Finally, if you set INCLUDEWEIGHTS to TRUE, then also call setInputEdgeWeights(), sending it the Tensor which contains the input edge weights.
At this point, the bean should begin to calculate automatically.

<table>
<thead>
<tr>
<th>property</th>
<th>input data type</th>
<th>req’d</th>
<th>default</th>
<th>Studio conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputCoordinates</td>
<td>Tensor</td>
<td>Y</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>InputEdgeWeights</td>
<td>Tensor</td>
<td>Y/N†</td>
<td>null</td>
<td>Y</td>
</tr>
<tr>
<td>ColVertexLabel</td>
<td>String</td>
<td>Y</td>
<td>&quot;events&quot;</td>
<td>N</td>
</tr>
<tr>
<td>EdgeLabel</td>
<td>String</td>
<td>Y</td>
<td>&quot;ties&quot;</td>
<td>N</td>
</tr>
<tr>
<td>IncludeWeights</td>
<td>boolean</td>
<td>Y</td>
<td>FALSE</td>
<td>N</td>
</tr>
<tr>
<td>RowVertexLabel</td>
<td>String</td>
<td>Y</td>
<td>&quot;actors&quot;</td>
<td>N</td>
</tr>
<tr>
<td>LogLevel</td>
<td>Level</td>
<td>Y</td>
<td>OFF</td>
<td>N</td>
</tr>
<tr>
<td>AutoCalculate</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
<tr>
<td>AutoClearInputs</td>
<td>boolean</td>
<td>Y</td>
<td>TRUE</td>
<td>N</td>
</tr>
</tbody>
</table>

†Y if IncludeWeights = TRUE, else N.

Table 6.31: Input Properties for TensorToGraphArray (T2GA)

Once calculation has finished, call back to getOutputGraphArray() to get the output GraphArray produced by this bean.

TensorToGraphArray is employed in the 2-mode tutorial in Section 4.4.1.1 and in the 1-mode tutorial in Section 5.9.1.
Table 6.32: Callback Properties for `TensorToGraphArray` (T2GA)

<table>
<thead>
<tr>
<th>property</th>
<th>output data type</th>
<th>Studio callback</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OUTPUTGRAPHARRAY</code></td>
<td><code>GraphArray</code></td>
<td>Y</td>
</tr>
<tr>
<td><code>COLVERTEXLABEL</code></td>
<td><code>String</code></td>
<td>N</td>
</tr>
<tr>
<td><code>EDGELABEL</code></td>
<td><code>String</code></td>
<td>N</td>
</tr>
<tr>
<td><code>INCLUDEWEIGHTS</code></td>
<td><code>boolean</code></td>
<td>N</td>
</tr>
<tr>
<td><code>ROWVERTEXLABEL</code></td>
<td><code>String</code></td>
<td>N</td>
</tr>
</tbody>
</table>
CHAPTER 7

Programming Guide

7.1 Introduction

The purpose of this chapter is to provide some general guidelines for writing FunctionBean-descendant Java beans. It is assumed that the programmer reading this chapter is familiar with Java, the basic concepts of object-oriented programming, and with exception-handling.

First the chapter covers the basic background information necessary when programming a FunctionBean-based bean. Second, the basic steps in programming such a bean are covered, with examples from my own beans. Third, information on optionally extending your bean with a BeanInfo class is given. Fourth, basic guidelines for packing up your bean or beans for delivery to the Studio environment are given. Fifth, a basic example is given of using a FunctionBean-based bean in a programmatic environment. Finally, information which will be helpful to programmers wishing to use the Tensor and GraphArray data types is provided.

The information contained in this chapter is designed to be helpful to the programmer but is secondary to consulting the source code. These will be available at the socnet beans website, http://orion.oac.uci.edu/~jastern/socnetbeans/
7.2 Programming a Java Bean

In creating a FunctionBean-based bean, it is possible to simply start with the code for one of my beans, altering it for your bean’s particular use. This should make the process somewhat easier. Nonetheless, it will still help you to write better code if you understand the basics of writing a Java bean, as well. This includes understanding properties and their accessor methods (getters and setters), synchronization and multi-threading, serialization and persistence (and the new long-term persistence rules for Java beans – see Section 7.4.5.1 below).

There are helpful books available. Two that I used were Developing Java Beans (Englander 1997) and JavaBeans Unleashed (Doherty and Leinecker 2000). There are others listed at Java Beans documentation web site (Sun Microsystems Inc 2003f), as well as helpful online tutorials including one by Quinn (2003), mentioned in Section 4.

That said, one of the main points (c.f. Section 11) of this dissertation project was to remove much of the responsibility and house-keeping of Java beans so that researchers could concentrate on their bean’s functionality in a data-flow environment.

Therefore, the following general guidelines are intended to be helpful for a wide range of Java programmers, from the beginner to the expert. Hopefully, with these simple rules, a person with fundamental Java experience will be able to make a FunctionBean-descendant Java Bean which will work together with other FunctionBeans in GeoVista Studio and other environments.

These instructions are most likely far from complete, and if there is sufficient interest after the dissertation, I will improve them based upon user experiences. Please check the website (http://orion.oac.uci.edu/~jastern/).
7.3 Background to Programming a FunctionBean

Before explaining in detail how to program a bean which extends FunctionBean, it is important to outline the class hierarchy of the FunctionBean (and thus all socnet beans provided as proof-of-concept in this dissertation). It will also be useful if the reader understands FunctionBean’s parent, StackLogger, and how it may be used to conveniently log events during the bean’s functioning. Finally, the basics of thread-safe coding and synchronization, as they relate to a FunctionBean will be covered.

7.3.1 Class hierarchy of the Socnet beans

Table 7.1 lists the full package names of the 15 Socnet beans created for this project.

Figure 7.1 depicts a simplified UML class hierarchy of these beans. Each of the 15 Socnet beans (such as GraphArrayToMage, or ReadDL) descends from FunctionBean, which descends from StackLogger. This means that all the Socnet beans (and any bean you extend from FunctionBean) have all the functionality of a FunctionBean and of a StackLogger.

Figure 7.2 shows a UML (Booch et al., 1998) class hierarchy of one of the Socnet beans, ReWeight. The figure shows that FunctionBean extends StackLogger (and the java.io.Serializable interface), and that ReWeight extends FunctionBean. The details of this diagram are not important. However, notice that many of the methods available to ReWeight are inherited from FunctionBean and from StackLogger. These are methods you would otherwise have to
- socnet.jastern.ct.CollapseTensor
- socnet.jastern.dl.ReadDL
- socnet.jastern.et.ExpandTensor
- socnet.jastern.ga2m.GraphArray2Mage
- socnet.jastern.ga2pdb.GraphArray2PDB
- socnet.jastern.os.OptScale
- socnet.jastern.ots.ObjectToString
- socnet.jastern.rt.ReverseTensor
- socnet.jastern.rw.ReWeight
- socnet.jastern.srn.SquareRootNorm
- socnet.jastern.stf.StringToFile
- socnet.jastern.stkt.StackTensors
- socnet.jastern.svd.SVD
- socnet.jastern.sym.SymmetrizeTensor
- socnet.jastern.t2ga.TensorToGraphArray

Table 7.1: The socnet.jastern sub-packages

program if you were starting a data-flow bean from scratch, but which instead are available to you when you extend FunctionBean.

The use of the method calls from StackLogger and FunctionBean will be
7.3.2 Logging events with StackLogger

All FunctionBean-extended beans also descend from socnet.core.StackLogger, and so have its ability to log their activities.

StackLogger enhances the facilities of the java.util.logging.Logger and java.util.logging.ConsoleHandler classes, simultaneously making them easier and more convenient to use. For instance, the StackLogger calls also keep track of the current method that the logging call was made from. This way, the programmer gets the added benefit of knowing the method the logging entry was made from, without the additional typing for each logging call he/she makes. Instead, the programmer simply logs the entering of the method by name with...
Figure 7.2: ReWeight’s relationship to FunctionBean and StackLogger
an entering() call.

As covered in Section 6.1.4 there are seven logging levels — SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, as well as OFF and ALL. These logging levels are inclusive, meaning that if you set a logging level of INFO, you will see all INFO, WARNING, and SEVERE messages displayed. Thus, for most purposes, FINEST is equivalent to ALL.

The usual practice is that for any method you want to do logging on, you put an entering() StackLogger call at the beginning of, before any other code in that method. The entering() call contains the name of the method you have just entered, such as:

```java
protected synchronized boolean calculate() {
    entering("calculate");
    ...
}
```

Then you put an exiting() logging call at the end of the method. If you are returning a value from this method, you put the exiting() call right before it, and include the value of the returned variable, as well:

```java
    ...
    exiting(bRetVal);
    return bRetVal;
}
```

Whenever making an entering() call, care must be taken to accompany it with an exiting() call. This is because the StackLogger keeps a paired stack of the called methods internally, so that when another logging call is made from within the method, the StackLogger can log it along with the current name of the method that the bean is running in. The only way it knows this is by the entering() call which was made at the beginning of the method. Therefore, if the programmer
forgets one side of the `entering()/exiting()` pair, the logs that are created will be incorrect. An exception may also be thrown.

`entering()` and `exiting()` each have more than one way to call them. `entering()` is always called with a `String` which is the name of the method just entered. An example of this may be seen in `GraphArrayToMage.readyToCalculate()` in Figure 7.12. First an `entering()` call is made, along with the `String` value of the name of the method itself — `readyToCalculate`. At the end of the method, an `exiting()` call is made, and along with it is sent a copy of the return value itself. This copy may be an `int`, a `boolean`, or an `Object` of any kind. This return value should be a `copy`, not the value itself.

The other two versions of `entering()` (as seen in Figure 7.2) are for methods which have one argument (in which case the `entering(String, Object)` version is used), or have more than one input argument (in which case an array is assembled and the `entering(String, Object[])` version is used).

`entering()` and `exiting()` log their events at the level of FINER. Generally, for the socnet beans, I used a level of FINEST for events I only wanted to see while debugging, WARNING for events which should be brought to the attention of the user, but which may not stop execution, and SEVERE for events which stop the execution of the bean because of a fatal problem.

Events of any level may be logged with corresponding `StackLogger` calls: `severe()`, `warning()`, `info()`, `config()`, `fine`, `finer`, and `finest`. These calls — unlike `entering()` — do not need to have the method name included in order to have the method they were called from logged. This is because the `StackLogger` stack remembers the last `entering()` call which was made, and which method it was called from. Therefore, when making any of the other calls, the method the call was made from will be included in the text of the log automatically.
Any bean descending from StackLogger also inherits the calls setLogLevel and getLogLevel(), which may be used to set the bean’s current logging level. As mentioned in Section 7.4.12.1, setLogLevel() may be overridden in the bean itself in order to also set the logging level of any contained objects (fields) at the same time.

Finally, when throwing an exception (or passing one up), one may use the throwing() call, which will log (at a level of SEVERE), that an exception is about to be thrown. One should use this immediately preceding the throwing of an exception. This may be quite helpful during debugging, since fastidious use of this call will help the StackLogger facilities to get you more information about where the exception was thrown than is always in an exception stack.

7.3.3 The Data Processing Cycle of a FunctionBean

To program beans which descend from FunctionBean, it is important to understand the basic data processing cycle of FunctionBean. Figure 7.3 shows the basic outline of FunctionBean’s data processing cycle. There is more involved to it than this of course, but this is the basic outline you will need to understand how to program a descendant of FunctionBean.

First an input property is set. Next, the bean determines whether it has been set for autocalculation. This means it will automatically begin to calculate once it has all the needed input properties. If it is set for auto-calculation, then the bean checks to see whether it is ready to calculate. If it is, it sends out a signal (a propertyChange event) for the benefit of any interested parties that it has begun calculating. Then it begins calculating its core functionality. After calculation is finished, the bean automatically clears its inputs if it has been set to do so. Then it sends a signal (again, a propertyChange event) that it has completed
its calculations. If calculation was not successful, it logs the fact. If it was, it sends out another signal (propertyChange event) that it has successfully finished calculating. At that point, any down-stream beans may call back to one of the bean’s callback methods to get one of its output properties (the data the bean created as part of its calculation).  

All of the above may be seen in Figure 7.3. Most of these actions are contained in FunctionBean’s doCalculate(), a method you do not have to override. Thus, most of the housekeeping is done for you, especially the signalling with propertyChange events and the like. The way this process boils down in terms of your methods may be seen in Figure 7.4. All of the methods in that figure are declared in FunctionBean. However, only those methods depicted in the figure with solid boxes represent methods that must be overridden in your bean, to implement your particular bean’s personality. The rest is taken care of by FunctionBean. Section 7.4 will outline the process of creating these methods for your bean.

7.3.4 Multi-threading and synchronization

In Studio multi-threading is not an issue for your bean because Studio creates a new instantiation of each bean you put on the DesignBox. Furthermore, FunctionBean is push-driven, meaning that no downstream bean in Studio will call back to an upstream bean for output data until it has received an output propertyChange.event even from the upstream bean, notifying the downstream bean that it has finished calculation.

However, in environments outside Studio, and especially in programmatic

---

1The exception to this is that if the bean is a terminating bean and it is only writing to a file, then no beans will call back for output data.
Figure 7.3: **FunctionBean**'s basic data processing cycle
Figure 7.4: Methods used in the **FunctionBean**’s data processing cycle
environments, many of these assumptions may not necessarily hold true.

The basic design for FunctionBean which keeps it thread-safe is that all the publicly-accessible methods (not just accessor methods) are declared to be synchronized. This will prevent another call back to any of these methods of your bean until the current program-counter (thread) which has already entered one of these synchronized methods exits.

7.4 Making your FunctionBean

7.4.1 Set up core file, declare package, imports, and Class

First, create the blank .java file you will need for your bean. To do this, you will have to decide first what your package tree will be.

For instance, all of the Socnet beans are in the socnet.jastern package tree. You may like this scheme and put yours under socnet.<yourname>, or you may choose a different scheme.\(^2\)

Your directory structure at the time you compile will have to reflect the package naming scheme you use. Thus, for my packages, I had to create a directory called socnet and within that, a subdirectory called jastern. For each bean, I created a folder underneath the jastern directory, as well.

I also create a separate subfolder for my bean, under the socnet/jastern directory. This will put this bean into a separate package from the other beans. The reason for doing is that when changing the LOGLEVEL of any object, the LOGLEVEL of all other objects in the same package will also be changed. (This is an inherent feature of the java.util.logging.Logger class.) Therefore, since

\(^2\)There is some information on package-naming conventions in Section 9 of Sun Microsystems Inc [1999]
users will usually want the ability to specify logging level on a per-bean basis, we will want to put each bean into its own package.

For instance, with my GraphArrayToMage bean, I created a socnet/jastern/ga2m directory, in which I created a GraphArrayToMage.java file. This puts GraphArrayToMage into its own package.

The first few lines of GraphArrayToMage.java look like this:

```java
// i always declare my package
package socnet.jastern.ga2m;

// for FunctionBean, StackLogger, and datatypes
import socnet.core.*;

// i have to import this because i am going to be using the
// java.util.logging.Level object explicitly
import java.util.logging.*;

// so i can fire java.beans.propertyChange’s explicitly
import java.beans.*;

// jdsl stuff
import jdsl.core.api.*;
import jdsl.core.ref.*;
import jdsl.graph.api.*;
import jdsl.graph.ref.*;

public class extends FunctionBean {
...
```

Figure 7.5: Starting the code for GraphArrayToMage

Notice that I specify the package, which matches the subdirectory tree I am in. I also import socnet.core.*, which includes all the support files, including FunctionBean’s definition, StackLogger, the Tensor and GraphArray datatypes, and so on.
I also import `java.util.logging.Level`, since in this bean I will be making explicity references to which `Level` explicitly in this bean, later on.

Finally I declare the class, extending `FunctionBean`, so as to get all its functionality.

### 7.4.2 Plan your input and output properties

The first real architectural or design step in setting up your `FunctionBean` descendant-bean is to consider what your bean is going to need in order to do its function — its input properties. Also, what will your bean’s output (callback) properties be? And what will the data format of each of these properties be?

In the simplest case, you might have an input `Tensor` and an output `Tensor`, as in the case of the `ReverseTensor` Socnet bean (see Section 6.2.8). In a more complex bean such as `ReWeight` (Section 6.2.9) or `SVD` (Section 6.2.13), you may have several input or output properties.

Carefully name these properties: Their naming will help the user understand what the property is, and how it is used. For instance, in programming the Socnet beans, I have employed the mechanism of using the word “input” or “output” in the name of the property, to help the user to understand how this property is being used by the bean.

Thus, if your property’s data type is an input `Tensor` which contains the scores for a midterm examination and your bean takes the average, you might call the input property, “INPUTSCORES”, or, if you prefer to reference just the data type (though this is less informational), “INPUTTENSOR”. Similarly with the output properties. Yours might be called “OUTPUTAVERAGEDSCORES” or perhaps just “OUTPUTTENSOR”.

265
Make a list of these properties, and name them.

### 7.4.3 Declare your private variables

Once you have planned what you will need for input and output properties, you will have to provide your bean its own private, internal variables for working with them.

In a simple bean like `ReverseTensor`, declaring fields will be a simple matter of declaring one variable for each property, as seen in Figure 7.6 (ignore for now the `serialVersionUID` field).

```java
   // FIELDS========================================================

   /* for future-version compatibility */
   // static final long serialVersionUID =

   /* input Tensor to be reversed */
   private Tensor tnsIn;

   /* output Tensor, reversed */
   private Tensor tnsOut;

   Figure 7.6: Declaring the fields for ReverseTensor
```

On the other hand, for a complex bean such as `GraphArrayToMage`, there will be several fields, each with a different purpose, as seen in Figure 7.7.

Since `ReverseTensor` is a simple bean requiring only one input property, `INPUTTENSOR` and one output property, `OUTPUTTENSOR`, I have two references to `Tensors` declared as fields. Their objects are not yet instantiated, but the references are ready.

Also, notice that with both beans there is a `serialVersionUID` field, which
Figure 7.7: Declaring the fields for `GraphArrayToMage`
at this early point in your code you would not have, but which will be referenced in Section 7.4.13. Actually, at this early point in the coding, I usually create the line containing the serialVersionUID variable, but comment it out until later, when I have the full code written for the bean.

More complex beans with more inputs and outputs will obviously require for private fields to do the job. It is up to you as a programmer. It is not the purpose of this chapter to go into depth with any one bean — only to outline the process. However, perusing the code in Figure 7.7 you may see how many of the input and callback properties listed in Tables 6.6 and 6.7 are implemented internally in this bean.

It is good practice to make all your variables private. Making these variables private helps to ensure that they will not be accessed directly by other objects. You want your bean to be accessed only by accessor methods — the getter and setter methods for the various input and output properties you have set up. These methods will be covered in the next section.

### 7.4.4 Program your setters and getters

For each input property you will have to program both a setter and a getter. The setter is the most important of these, since this is what the user uses to set the input. The getter is used only by the Studio (or other) environment to implement long-term persistence.
public synchronized void setInputGraphArray(GraphArray ga) {
    // log our entry into this method/procedure
    entering("setInputGraphArray", ga);

    // save reference to last value
    GraphArray gaOld = gaIn;

    // store new value
    gaIn = ga;

    // tell anyone who’s interested that we’ve made the change
    pcs.firePropertyChange("InputGraphArray", gaOld, gaIn);

    // run our input GraphArray at the same LogLevel we (the
    // bean) are running at
    if (gaIn != null)
        gaIn.setLogLevel(getLogLevel());

    // run if we’re ready
    autoCalculateIfSet();

    // log that we are leaving this method/procedure
    exiting();
}

Figure 7.8: Programming the setter for the InputGraphArray property

7.4.4.1 Input properties

Figure 7.8 shows the code to set the input InputGraphArray property. There are several things to notice in programming setters which this method demonstrates:

1. Log the entering of your setter.

   Use the StackLogger methods (which your bean inherits through Func-
tionBean), entering() and exiting() to log that this setter is being executed. That way if there are any errors, it may be more easily tracked. Also be sure with both entering() and exiting() to use the versions of these which allow you to declare input parameters (as you see in entering()) and returned variables (since we do not return anything with this setter, we use a zero-parameter version of exiting()).

2. Fire a property change for the setting of the property

There may be other beans or objects which have registered themselves with your bean (through the PropertyChangeSupport of FunctionBean) to be notified upon changes in your bean. This is a nice way to let the world know that something has happened. For instance, a GUI like Studio may wish to blink its icon (in a future version) when an input property is set. Firing this event when you set the property would be the perfect way to let Studio know this has happened.

To fire the property change, record the old value and send it along with the new value.

3. Set LogLevel of input properties

If your input parameter’s datatype itself descends from StackLogger — as do Tensor and GraphArray both — then you will usually want to automatically set the logging level of that object to the same logging level as your bean is running in, so that you see messages from both your bean and its contained objects. So when an input object is given for a property, a nice way to handle this is to automatically set that object’s LogLevel property to the same Level as your bean is currently running in.

4. Run autoCalculateIfSet()
Finally, the last thing you should do in your setter before calling `exiting()` is to call `autoCalculateIfSet()`. This method is inherited by your bean from `FunctionBean`, and checks whether your bean is set to autocalculate, and if so, starts the process of checking whether everything is ready and then calculating.

5. Log the exiting of your setter

Finally, log the exiting of your setter. If your setter by any chance returns something (though usually setters do not), then call `exiting()` with an argument: the object it returns. Also, call `exiting()` before you call `return`.

Programming the `getter` for your input property will usually be much simpler. Why should you program a `getter` for an `input` property? It is not necessary for using your beans in the Studio DesignBox, since `FunctionBeans` normally clear their inputs after calculation, anyway. But programming these extra accessor methods can be helpful in some cases for environments outside Studio, and for serialization and long-term persistence. It makes sense to make your bean work in as many environments as possible, and one of the best ways of doing this is making it long-term persistent.

```java
public synchronized GraphArray getInputGraphArray() {
    entering("getInputGraphArray");
    // make a copy to give away: we don’t want to give others
    // the ability to change our internal data objects
    GraphArray gaClone = In.clone();
    // now simply log it and return it
    exiting(gaClone);
    return gaClone;
}
```

Figure 7.9: Programming the getter for the `INPUTGRAPHARRAY` input property
An example of programming the getter for an input property is shown in Figure 7.9. Notice that we do not have to fire any property change. Also notice that for safety’s sake, it is probably better to make a clone of the property than to return a reference to it. This prevents our internal data (in this case the input GraphArray) from being changed by other objects without our bean knowing it.

Note the convention of having properties which are used for input have "Input" in their name. Normal Java beans do not have this convention, because they are not FunctionBeans. There is nothing programmatic which depends on this convention, so you have the power to violate this convention if you like. But following it is helpful to the user, especially when certain output properties are really inputs. An example of this is that you may want to set the name of a file which will be part of the output of your bean. But the name of this output file is actually — if you think about it carefully — an input property. The name of this output file is set as an input. The file itself is output. Therefore, the property should be called something along the lines of INPUTOUTFILENAME, and its accessor methods will be setInputOutFileName and getInputOutFileName.

7.4.4.2 Output properties

The programming of output properties is much the same as for input properties — just reversed. You program a getter, and an optional setter for each output property, as well.

Figure 7.10 shows the simplicity of a getter and its concomitant setter for GraphArrayToMage’s OUTPUTKINEMAGE output String property. Again, setters for output properties are written solely for long-term persistence compliance.
public synchronized String getOutputKinemage() {
    entering("getOutputKinemage");
    exiting(strKinOut);
    return strKinOut;
}

// for long term persistence only
public synchronized void setOutputKinemage(String km_out) {
    entering("setOutputKinemage");
    strKinOut = km_out;
    exiting();
}

Figure 7.10: Programming the accessor methods for the OutputKinemage output property

Notice the convention I use is to have ”Output” in the name of the property, so as to cue the user of the Bean Adapter Wizard in Studio, or programmers using the bean, that this is an output property.

7.4.5 Create a zero-parameter constructor

Next, you will want to create one or more constructors for your bean. There may be more than one of these, depending upon whether and how you will want to send various parameters to instantiate a new bean differently.

However, for long-term persistence, containing environments (such as Studio) must be able to re-instantiate a bean that was ”alive” on a previous use of a program. Therefore they must be able to create automatically a new running version of your bean without knowing any parameters a priori. This simply requires the addition of a zero-parameter constructor.

You may put any code you like within the constructor, though you should call super() first, since your bean descends from FunctionBean. Calling super()
ensures that FunctionBean’s constructor, and any code within it, gets called before anything else you do. Figure 7.11 shows GraphArrayToMage’s zero-parameter constructor.

```java
// CONSTRUCTORS===============================================

/** Creates new TensorToGraphArray */
public GraphArrayToMage() {
    super();
}
```

Figure 7.11: Declaring the zero-parameter constructor for GraphArrayToMage parameter constructor.

7.4.5.1 Supporting Long-term Persistence

If you are interested in supporting long-term persistence for your bean, then any data type that you use an input or output property should also support long-term persistence. Again, that simply means that the class should have a zero-parameter constructor (though it may have others as well), and that there should be setters and getters for all properties. This is so that an active bean’s state may be re-created at a later date.

Since FunctionBean implements java.io.Serializable, your bean does, too, with some caveats. You will need to override Java.io.writeObject and Java.io.readObject if you want to control the serialization process directly. Also, you should declare as transient any fields which will not need to be saved at the time of serialization. Most input and output properties (those receiving data, such as Tensors, Arrays and such) fall into this category. Any fields you would want to save (such as STARTING BALL COLOR or BALL COLORING METHOD) to a serialized bean should be serialized.

274
This may be implemented with a long-term persistence scheme. For more information on long-term persistence, see (Milne and Walrath, 2003) (Milne, 2003a) and articles on persistence and XML on The Swing Connection web site (Milne 2003b).

7.4.6 Override readyToCalculate()

Here is where you check that everything is ready. This method is called by doCalculate() (which itself is called by autoCalculateIfSet()). Generally this happens either after an input property has just been set, or after doCalculate() has been called manually.

This method returns TRUE if the bean is ready to calculate, otherwise it returns FALSE. You must return one of these values.

Being ready to calculate means that your bean reviews its state of readiness in three ways. These three checks are the purpose of the readyToCalculate() method, and therefore what you should undertake to program. Returning TRUE means that readyToCalculate() has:

1. Checked that all required inputs are non-null

Before your bean can go through one data-processing cycle, it must know that all the input properties it requires to run have been set. Here is where you check that.

In Figure 7.12, you can see that the INPUTGRAPHARRAY property (that is, the input GraphArray object) is checked to see whether it is non-null. If it is null, that means that this property has not been set yet, and that it is not yet time to run the bean. If it has been set, the code continues down further.
protected synchronized boolean readyToCalculate() {
    entering("readyToCalculate");

    // default return value
    boolean bRetVal = false;

    try {
        // is the input GraphArray non-null?
        if (gaIn == null) {
            finer("not ready: no input GraphArray set yet");
            exiting(new Boolean(false));
            return false;
        }

        // if we got this far, it means we passed all the tests, and
        // also had no exceptions, so return true to show we’re
        // ready to calculate.
        bRetVal = true;
    }
    catch (Exception e) {
        severe("Could not finish. "
            + "An exception was thrown which reads:"
            + strEOL
            + e);
        bRetVal = false;
    }

    exiting(new Boolean(bRetVal));
    return bRetVal;
}

Figure 7.12: Overriding GraphArrayToMage.readyToCalculate()
try {

    // input coordinates exist?
    if (tnsInCoords == null) {
        finer("not ready: no input coordinates set yet");
        exiting(new Boolean(false) );
        return false;
    }

    // input coordinates at least 2 columns (for 2-d coordinates. 
    // we’ll ignore 4th column and beyond)
    if (tnsInCoords.getNumCols() < 2) {
        severe("not ready: input coordinates tensor has less than 2" 
        + "columns. 
        + "need at least 2 columns to produce 2-D coordinates.");
        exiting(new Boolean(false) );
        return false;
    }

    // coords: at least 1 row
    if (tnsInCoords.getNumRows() < 1) {
        severe("not ready: input coordinates tensor needs at least" 
        + " 1 row.");
        exiting(new Boolean(false) );
        return false;
    }

    // coords: at least 1 matrix
    if (tnsInCoords.getNumMats() < 1) {
        severe("not ready: input coordinates tensor needs at least" 
        + " 1 matrix.");
        exiting(new Boolean(false) );
        return false;
    }

    ...

Figure 7.13: Overriding TensorToGraphArray.readyToCalculate()
2. Checked that all conditional inputs are also non-null

Furthermore, some of your properties may or may not be required, depending upon the state of other properties in the same bean. Your `readyToCalculate()` method should be aware of these conditions and check only those properties which are currently needed.

For instance, the **ExpandTensor** bean has some conditional properties (c.f. Table [6.4]): If the input property, `REQUIREMATRIXLABELS` is set to `TRUE`, then the bean must also check for the existence of a `StringList` which contains the input property, `NEWMATRIXLABELS`. Therefore, the `readyToCalculate()` method of that bean must check that `NEWMATRIXLABELS` is set (and validate its contents) only if `REQUIREMATRIXLABELS` is `TRUE`, and so on.

3. Validated all non-null input properties

While the code in Figure [7.12] does not show it, this method may also be used to do basic validation on your input properties, if they have been set. Figure [7.13] is a snippet of code from the same section of `readyToCalculate()` from a different bean, **TensorToGraphArray**. Here you can see that after the input coordinates (which is a `Tensor`) have been determined to be non-null, the code goes on to check the input `Tensor`'s dimensions: it must have at least 2 columns, 1 row, and 1 matrix. If it is non-null, but nevertheless does not meet these criteria, then **TensorToGraphArray.readyToCalculate()** returns `FALSE`.

Also, if you have more than one input property, you may validate each one of them in this method.

Notice that logging is used here: When checking for the existence of any
object at all (that it is non-null), we log with a level of FINER, since it is not an error that the input property has not been set.

However, when validating a non-null property, if there are errors, it means we can go no further. This means a log at the level of SEVERE is legitimate (the user may not be interested in low-level messages, but data errors would be of interest and he or she would want to see the message).

While it is required that you check your input properties for existence in readyToCalculate(), it is possible that you could wait to validate them in your calculate() method, or one of its sub-methods. Indeed, certain data types are simply too complex to validate easily — GraphArray being one of these.

However, checking and validating your input properties here in readyToCalculate() means that the code you write in calculate() can assume — indeed, can count on — that the input properties are not only there, but that they are healthy and useable. Any assumptions you make in your calculate() method (or any methods it calls) should be based upon checking and validation you performed in readyToCalculate(). This makes your code in calculate() shorter and easier to read.

If readyToCalculate() returns TRUE, the bean knows that it may continue on to call calculate().

### 7.4.7 Override calculate()

Of the FunctionBean methods, this is the most central. calculate() is called to execute the primary functioning of your bean, once readyToCalculate() returns TRUE, indicating that your bean is ready to do its processing. All inputs have
protected synchronized boolean calculate() {
    entering("calculate");
    boolean bRetVal = false;
    try {

        // gaIn should exist (tested in readyToCalculate()).
        // create the new stringbuffer for temporary output
        sb = new StringBuffer();

        // and now create the kinemage
        writeKineMage();

        // if we get this far (w/no exception), then we were
        // successful, so set the output string
        strKinOut = sb.toString();

        // and set success flag
        bRetVal = true;
    }
    catch (Exception e) {
        severe("Could not finish. An exception was thrown which reads:
             + strEOL
             + e");
    }
    exiting(new Boolean(bRetVal));
    return bRetVal;
}

Figure 7.14: Overriding GraphArrayToMage.calculate()
been set and validated. It is time for the bean to do its primary function.

However, unless the functioning of the bean can be done in just a few lines, it is recommended to separate out from `calculate()` the actual core functioning of your bean into a `private` method of your own naming, just to make the code easier to read.

Such a design may be seen in Figure 7.14. Here the reader may see that the primary function of the `GraphArrayToMage` bean has been encapsulated in a separate method, `writeKineMage()`.

However, despite calling a `private` method like `writeKineMage()` which does the core processing, there are several things that your `calculate()` method should take care of. Therefore, in programming this method, you should check to make sure that you:

1. Declare your method correctly

   Your `calculate()` method should be declared as `protected`. It should be `protected` so that it cannot be accessed from other code except code declared to be in the same package or by subclasses. It should be `synchronized` so that if your bean is used in a multi-threaded environment, it will operate in a thread-safe manner (that is, none of its `synchronized` methods will be allowed to be accessed until `calculate()` has finished). And it should return a `boolean` back to `doCalculate()`, notifying that method of `calculate()`’s success in completing its duties.

2. Log events

   As you have seen earlier, you will want to log the entering and exiting of your `calculate()` procedure. Furthermore, you may want to mark milestones in the completion of the method with `finer()` or `finest` logging calls.
3. Trap exceptions

Also, you should not allow any exceptions (whether compiler- or run-time) to escape your calculate() procedure. Thus, the try...catch block you see in Figure 7.14. If you do get an exception, you should return FALSE so that your bean can notify the user of an unsuccessful completion of the calculate() method. Finally, you should log the exception with a severe() logging call. As you can see in Figure 7.14, my style is to call severe() and also pass it the string contents of the trapped exception, so as to give more information back to the user.

4. Create the fields for your output properties

Notice finally that calculate() is a good place to create your output variables (or fields, in object-oriented terminology). Notice in Figure 7.14 that sb (a field created globally to the bean at the top of the code) is assigned a new instance of a StringBuffer on each new run (The JVM garbage collector will take care of the old one.)

7.4.8 Write your core code

This is the code you call from within calculate. For instance, in GraphArrayToMage, it is separated out into a private method called writeKineMage(). However, if the code is small enough, you may wish to include it within calculate().

Your core code may assume that all input variables have been validated in readyToCalculate() and output variables prepared in calculate(), so it may start right in on its work.

If you make this a separate method, declare it as private. The code may
return a boolean if you like, of course. It may also return exceptions, too (as long
as they are trapped by calculate()).

Also, log your code. As usual, employ entering() and exiting() to track the
use of a separate method. Also, use finest() to log sub-parts of your code which
you feel important-enough to track for debugging purposes, but which most users
would not be interested in.

7.4.9 Override clearInputs()

This method usually is called after calculate() has finished. It clears the input
properties in order to prepare the bean for another data processing cycle. The
point of clearing the input properties, from the standpoint of the bean, is so that
it will not re-use input properties which were set from earlier processing cycles.
Clearing them all out ensures that the bean will not be ready to calculate until
all input properties are re-set with new data.

    public synchronized void clearInputs() {
        entering("clearInputs");
        gaIn = null;
        sb = null;
        resetBallColorPointer();
        exiting();
    }

Figure 7.15: Overriding clearInputs() for GraphArrayToMage

An example may be seen in Figure [7.15] where the bean clears the fields gaIn
(for its INPUTGRAPHARRAY property) and sb (a StringBuffer used to hold
the output string for the OUTPUTKINEMAGE output property). In this case, the
bean is clearing not only its input properties, but also any other fields which must
be used during calculation.

Notice that it is sufficient to simply set these references to null, since the Java Virtual Machine’s garbage collection will recycle the memory the old objects occupied, and since the next invocation of GraphArrayToMage.calculate() will create new instances of these objects.

Also note that one should not make calls to any setters from within clearInputs(), since setting a property in a FunctionBean leads to a possible running of doCalculate(), and thus readyToCalculate() and calculate(), and back to clearInputs() – an invitation to an infinite loop. Instead, just set your private fields directly.

7.4.10 Decide on Autocalculation

Decide whether you want your bean to automatically calculate once all input properties have been set. With most beans, this property — AutoCalculate — would be set to TRUE. Thus, the default for any FunctionBean is TRUE.

```java
public ReadDL() {
    super();
    setAutoCalculate(false);
    setAutoClearInputs(false);
}
```

Figure 7.16: Turning off AutoCalculation and AutoClearing of Inputs for ReadDL

However, with certain beans — most notably those which initiate a data flow network — you will want to set this property to FALSE. This is because you may want to prevent the network running when all inputs have been set on the first, or initiating bean. That is, you want to run the network manually.
An example of this may be seen with the ReadDL bean. ReadDL has only one major input property — the name of the input file it will be reading from. As seen in the tutorials in Chapters 4 and 5 when ReadDL is used, it usually initiates a dataflow pipeline. This is where the data starts in a data flow network. Thus, we want the default value of AutoCalculate to be FALSE for this bean. We want to start the bean manually. To do so call the bean’s doCalculate() method, usually with a button or some other such GUI-based signalling device.

Figure 7.16 shows where the programming of the default for this property would normally take place — in the constructor(s) for the bean itself. Simply calling setAutoCalculate() with an argument of false will do the job.

Of course, this may always be overridden later by the user, in Studio, should he or she wish to auto-calculate when the name of the input file has been set. However, for most users, the default — to do nothing — would be the expected behavior, and setting the property this way will ensure your initiating bean does not fly out of the gate before the gun is fired.

It should be noted however, that even a normally initiating bean such as ReadDL could be in the middle of a data flow network under other circumstances. As an example, it would be possible to set up a data flow network to read a series of input files, creating a series of output files in the process. Thus, the network could be set up to automatically process several, or many input DL-formatted files, in batch style. An initiator bean (programmed later, not existing now) could read in a list of file names and send them one-by-one to ReadDL. In that case, it would be necessary to simply set ReadDL’s AutoCalculate property to TRUE, so that the bean would automatically read in a DL file each time a new file name were sent to it (i.e., each time its INPUTFILENAME property were set).
7.4.11 Decide on AutoClearing of Inputs

The reader may also notice in Figure 7.16 that a second property has been set in ReadDL’s constructor, as well – AutoClearInputs.

Once again, for most beans, we will want to automatically clear all input properties at the end of a data processing cycle. Thus, the default for any bean extending FunctionBean is that AutoClearInputs is TRUE.

As with auto-calculation, there are exceptions to auto-clearing. At times, you will want your bean to not clear inputs between data processing cycles. This is usually for beans which reside at the beginning or end of a dataflow network.

In the case of ReadDL, we do not want to clear all input properties. ReadDL has a single non-inherited input property — InputFileName. This property would not normally be cleared at the end of a data processing cycle. If it were, a second data run would not be possible, since now the bean would not know which file to read in.

The same is true of terminating beans which may write to output files: the preferred behavior will be for the bean to remember the name of the output file.

Therefore, beans such as ReadDL (normally an initiator bean in a dataflow network) and StringToFile (normally a terminating bean) set their AutoClearInputs property to FALSE by default, in their constructors. Note, however, that terminating beans still leave their AutoCalculate property set to TRUE.

There is some room for design discretion here. First, even with the defaults set in the constructor of these beans, users and programmers using the beans may still set the values of these properties back. Second, another choice might be to program the clearInputs() method (Section 7.4.9) so as to not clear all input
or output properties. Thus, you could leave AutoClearInputs set to TRUE, knowing that when clearInputs() is called, not all inputs will be cleared.

However, it is probably better programming practice to clear all inputs with clearInputs(), and leave individual instances up to the user.\footnote{Indeed, in a future version, with sufficient interest, the author (or other contributors) may enhance FunctionBean so that it be possible for the user to specify which inputs to auto-clear, not just all or none}

7.4.12 Decide on Logging Level

Logging level is normally OFF for all beans though this may be overridden in the normal way by the user of Studio who is accessing the bean’s properties. Your bean’s logging level may also be saved in a different state during serialization.

7.4.12.1 Override setLogLevel()

You may decide that you would like to override setLogLevel() as well, though this is optional and will not affect the core functioning of the bean if you omit it.

The purpose of overriding setLogLevel() would normally be to take the opportunity to also set any objects your bean owns to a similar logging level. This would only work, of course, for objects which are descendants of StackLogger. Two of the datatypes used by the socnet beans are Tensor and GraphArray. These both descend from StackLogger. Thus, normally when you set the log level of your bean, you might want to automatically move the logging level of any of your bean’s contained StackLogger-derived objects (fields) to the same logging level.

An example of this may be seen in Figure \ref{fig:7.17}. First, a call to super() is made in order to call FunctionBean’s setLogLevel() method (which itself calls
public void setLogLevel(Level newlevel) {
    super.setLogLevel(newlevel);
    if (gaIn != null)
        gaIn.setLogLevel(newlevel);
}

Figure 7.17: Overriding setLogLevel() for GraphArrayToMage

StackLogger’s setLogLevel()). Then the input GraphArray’s logging level is set to the same level the bean’s logging level has just been set to. This way, a user who has just set the bean’s log level from OFF to CONST or FINEST, will have set the bean’s input GraphArray’s logging to the same level. This will allow the user to see everything that is going on in the bean.

7.4.13 Add an SUID: a Stream Unique Identifier

Once you have gotten your bean sufficiently along in development, you will want to use the serialver shell program (which comes with the J2SDK) to create an SUID, or Stream Unique Identifier. Normally the JVM creates one for any bean which does not create one for itself. However, you can have your bean create one for itself, by reference to a serialVersionUID variable — a long integer — in your bean’s class definition. You can create a value for this variable using the serialver program which comes with the Java2 SDK.

The Stream Unique Identifier, or SUID, is used by the JVM to determine the compatibility of your bean with other software currently running. When applets and applications are compiled, the SUID is saved as part of the application. Should the user upgrade their version of your bean, the new bean will not automatically work with the user’s programs — even if you have been careful to make your newer version backward-compatible, in terms of its API. Instead, they will
receive a rather unpleasant `java.io.InvalidClassException` exception.

Therefore, to indicate to the user’s JVM that your newer version is still backward-compatible, your newer bean would use the same SUID that your earlier version did. Conversely, for newer versions of your bean which are *not* backward-compatible, you would use a new SUID for the newer beans — and keep using it for subsequent versions in that line.

The process of generating an SUID is relatively simply. From the shell prompt, change directory to the top of the package tree for your bean. Then run `serialver` on the full package name of your bean (without `.java` or `.class` at the end). Thus for the `GraphArrayToMage` bean, we `cd` to the directory *above* the `socnet` directory, then run `serialver` on `socnet.jastern.ga2m.GraphArrayToMage`, as seen in Figure 7.18 Then we copy the output line which begins with "`static final..`" into the declaration section of the `GraphArrayToMage.java` file, as seen in Figure 7.19

```
$ serialver socnet.jastern.ga2m.GraphArrayToMage
socnet.jastern.ga2m.GraphArrayToMage:
    static final long serialVersionUID = -3701435792345323534L;
$ 
```

Figure 7.18: Running `serialver`

The only reason for manually generating the SUID yourself is that if you did not, it would be automatically created for you by the JVM for each of the versions of your beans. If you added a new method to your bean, a new SUID would be generated automatically by the JVM the bean was running under, and your user would have the problems mentioned at the beginning of this section. Setting it yourself manually means that you retain control over the versioning.
public class GraphArrayToMage extends FunctionBean {

    // FIELDS=""""""

    /** for future version compatibility */
    static final long serialVersionUID = -3701435792345323534L;
    ...

Figure 7.19: Pasting in serialver's output

See Englander (1997, p.115-117) and/or Doherty and Leinecker (2000, 79-81)
for deeper discussions of this issue.

7.5 Extending your bean’s functionality with a BeanInfo class

At this point you may go ahead and use the bean as is, and it will work not only
as a Java bean but also as a FunctionBean.

However, you may wish to add one or more features to your bean. The
following sections outline the process of adding unique icons to your beans, for
use in visual environment such as Studio, and adding Customizers and Editors
for data validation, when setting input properties on your bean. While these are
optional additions, I have included them on all 15 of the beans programmed for
this project because they do make using the bean easier. The general process is
to make whatever you need (an icon, and editor, etc.) and then register it with
a separate BeanInfo class to accompany your own.
7.5.1 Create the bean’s icons

While an icon for your bean is optional, it is nice to create one so that it provides an easy visual cue to each bean when in Studio and other visual environments. According to the Java Beans specification, you may supply up to four versions of your bean: two each for black and white vs. color and for 32x32 pixels or 16x16.

Create the icons for your bean using whatever tool you choose. However, if you supply black and white bean icons, these should be monochrome (not greyscale).

7.5.2 Create any needed property editor classes

If you have input types which are beyond the normal primitives (integers, boolean, etc.) or the normal Java classes handleable by Studio (or other environment) such as String, then you will have to supply editors for each of these inputs, so that the available choices will be made available to the user.

Editors are excellent for data validation, where you have an input which may accept only a limited number of options.

For instance, with the Socnet bean, one of the input properties is BALLCOLORINGMETHOD, whose type is String. I could let Studio handle this automatically, since Studio will present String properties to the user with no objection.

However, there is a problem with doing so: Since there are only four ball coloring methods, I need a way to prevent the user of Studio from using anything other than ust those four methods. The methods are: ("INARIANT", "BY FRAME", "BY BALL", and "GRAPHARRAY COLORS"). For Studio to be able to know what the four legal values are so that it may present those to the user, it has to ask another object what they are. That object is an editor class you supply and
program for that property.

Thus, for the BallColoringMethod property of the GraphArrayTo-
Mage bean, I named the editor GA2MBallColoringMethodEditor, and its
code is in GA2MBallColoringMethodEditor.java, in the same directory (and
listed as in the same package) as .java. To make an editor for one of your bean’s
input properties, you may emulate this code with your own bean’s editor(s),
substituting your bean’s values.

It should be noted that there is nothing particularly different for Function-
Bean descendents, with respect to making editors.

You may have more than one editor of course, if you have more than one
input property whose values you need to control.

Sections of (Englander, 1997, pp.222-241) and (Doherty and Leinecker, 2000,
pp.537-540) tutorialize the development of editors for your bean.

7.5.3 Create any needed Customizer classes

Customizers may also be programmed for your Java Bean. A customizer goes
even further than an editor, in that it provides an entire window of its own which
contains some form of graphical input for your input property.

(Doherty and Leinecker, 2000, pp.540-547) provides an example of creating
a customizer for colors by simply calling the already-programmed, “built-in”
javax.swing.JColorChooser class that comes with available with your Java
compiler. Calling this class from within your own customizer is an easy way to
provide a color chooser the user can use with their mouse to easily choose a color
for a property without having to type in a name or an RGB or HSV value. The
customizer simply returns an object of type Color to the bean, and the bean’s
setter is called with this new value.

There are no **FunctionBeans** using a customizer as of this writing. However, their programming and declaration is similar to that of editors. Furthermore, the setting of colors *could* have been used in the **TensorToGraphArray** bean, or the **GraphArrayToMage** or **GraphArrayToPDB** beans, and may even be so employed in a future version. (Englander 1997, pp.241-252) creates a customizer with several controls on it, controlling not one, but several properties of his **Boiler** class.

### 7.5.4 Create a BeanInfo class

If you have created icons for your bean, or if you have created editors or customizers for one or more of the input properties of your bean, you will have to declare these in a BeanInfo class. The name of this class will have to start with the name of your bean and end with BeanInfo. For instance, the **ReWeight** bean’s BeanInfo class is **ReWeightBeanInfo** and its source code is in **ReWeightBeanInfo.java** in the same directory with **ReWeight.java**.

First, extend **SimpleBeanInfo** in your BeanInfo class declaration. This makes it easier to write the BeanInfo class because **SimpleBeanInfo** implements empty versions of a number of methods needed in a BeanInfo class: You only have to override the methods you need to. The following is the beginning few lines of **BeanInfo.java**:

This starts your BeanInfo file out. As just mentioned, extending **SimpleBeanInfo** (which takes care of creating skeleton versions of all the required methods below — and even ones we have not talked about) means you only have to write (override) the methods that you want to, depending upon whether you want to provide more information about your bean, and on whether you have
package socnet.jastern.ga2m;

import java.awt.Image;
import java.beans.*;
import java.lang.reflect.*;

public class BeanInfo extends SimpleBeanInfo {
    ...

    Figure 7.20: Starting the code for **BeanInfo**

created icons, and editors, and customizers for your various input properties.

Thus, you now write one or more methods detailed below, overriding **SimpleBeanInfo**'s skeleton methods, in order to provide more specific information for your bean:

### 7.5.4.1 Providing more info about your bean: Override **SimpleBeanInfo.getBeanDescriptor()**

Using **getBeanDescriptor()** you can set informational properties for the entire bean, some of which will appear in Studio and other graphical environments. These may also be queried in programmatic environments as well. Here you may set the bean’s display name, its long name, a short description, and any other values you would like. Notice that in my beans, I use **setValue()** to set an author (me) and a URL (a subdirectory of the socnet beans web site, [http://orion.oac.uci.edu/~jastern/socnetbeans/](http://orion.oac.uci.edu/~jastern/socnetbeans/)). (However, these two values will not be picked up by the current version of Studio.) Here is an example of providing a descriptor for the entire bean in **getBeanDescriptor** taken from **ReWeightBeanInfo.java**:

While this method is technically optional (as are all methods described here,
public BeanDescriptor getBeanDescriptor() {
    BeanDescriptor bd = new BeanDescriptor(.class);
    bd.setDisplayName("GA2M");
    bd.setName("GA2M");
    bd.setShortDescription("Creates a KineMage String Object from a GraphArray.");
    bd.setValue("author", "Jeff Stern");
    bd.setValue("url", "http://socbeans.org/beans/ga2m/");
    return bd;
}

Figure 7.21: Overriding getBeanDescriptor()

for the simplest beans), it is a good idea to write it, so that your bean behaves nicely (provides information about itself in Studio) and gives you credit, etc.

7.5.4.2 Making your icons show up in Studio: Override SimpleBeanInfo.getIcon()

getIcon() makes any graphical environment (such as Studio) aware of the icons available to represent your bean. If you have bean icons, you will need to write this method. Here is ReWeightBeanInfo.java's getIcon() method: Simply copy

public Image getIcon( int iconKind ) { Image image = null;
    if (iconKind == BeanInfo.ICON_COLOR_16x16) image = loadImage("ga2m_16x16_color.gif");
    else if (iconKind == BeanInfo.ICON_MONO_16x16) image = loadImage("ga2m_16x16_mono.gif");
    else if (iconKind == BeanInfo.ICON_COLOR_32x32) image = loadImage("ga2m_32x32_color.gif");
    else if (iconKind == BeanInfo.ICON_MONO_32x32) image = loadImage("ga2m_32x32_mono.gif");
    return image; }

Figure 7.22: Overriding getIcon()
this over to your bean and change the names to reflect the names of the four .gif files you created as icons for your beans.

7.5.4.3 Making editors and customizers for special input properties show up in Studio: Override SimpleBeanInfo.getPropertyDescriptors()

If you have programmed any editors or customizers, you will have to declare them here as part of the descriptors for each property. To do this, you override SimpleBeanInfo’s getPropertyDescriptors() method by adding this method to your BeanInfo java file:

```java
public PropertyDescriptor[] getPropertyDescriptors() {
    ...
}
```

Figure 7.23: Overriding getPropertyDescriptors(): starting

Please note that if you override this method to declare a property and an editor (or customizer) for it, then unfortunately, you will also have to make property descriptors for all other properties in your bean as well, even though you are not providing editors or customizers for those properties. This is because once you override the getPropertyDescriptors() method, then the Java Virtual Machine will use this information as the sole source of information about the properties of your bean. Thus it will refrain from attempting to discover your bean’s properties by itself (the latter process is called introspection). So, the rule is: declare one, declare all.

Here is the start of the getPropertyDescriptors() method of BeanInfo.java.
I specify two properties, and the editor for each one. I do it inside a try..catch block in order to catch any exceptions. Notice that there are three properties being set. Only two of these have editors attached to them:

Please note further that you must provide this additional information not only for properties in your own bean, but also for properties inherited by your bean from above, such as from `FunctionBean` and `StackLogger`. If you do not do this, then these properties — such as `LOGLEVEL`, `AUTO_CALCULATE` and `AUTO_CLEAR_INPUTS` — will not be presented to the user in Studio or other graphical environments. The code below continues the `getPropertyDescriptors()` method in `GraphArrayToMage.java`. It shows how these inherited properties are declared — just the same way the bean’s own properties are declared. Notice that along with declaring the already-inherited property comes the responsibility of also declaring the propertyEditor for it. Thus, the `LogLevelEditor` class must be declared along with the `LOGLEVEL` property itself:

Finally, at the end of `getPropertyDescriptors()`, all property-descriptors — both the bean’s own, and the inherited ones it had to ”re-describe” — must be registered and returned in an array, along with any ”attached” editors, to whoever asked for it (such as Studio). The try..catch block is closed. (Any exceptions just result in returning `null` to the caller – meaning no properties will be declared):

While this all sounds quite involved, it is not in the end as bad as it sounds, since for your own bean’s properties you can emulate my code, replacing the names of my properties with those from your bean, and regarding the properties from the ancestor classes `FunctionBean` and `StackLogger`, you can copy it over verbatim from the code above.
public PropertyDescriptor[] getPropertyDescriptors() {
    try {
        // create a descriptor for what starting ball color well use
        // ("SBC" = "StartingBallColor")
        PropertyDescriptor pdSBC =
            new PropertyDescriptor("startingBallColor",
                class);

        // now specify the propertyEditor for StartingBallColor
        // property descriptor
        pdSBC.setPropertyEditorClass(
            GA2MStartingBallColorEditor.class);

        // create a descriptor for what method of coloring balls
        // we will use ("BCM" = "BallColoringMethod")
        PropertyDescriptor pdBCM =
            new PropertyDescriptor("ballColoringMethod",
                class);

        // now specify the propertyEditor for ballColoringMethod
        // property descriptor
        pdBCM.setPropertyEditorClass(
            GA2MBallColoringMethodEditor.class);

        // same goes for other normal (don’t have special Editors)
        // properties in this object itself: still have to declare
        // them so they’ll be seen
        PropertyDescriptor pdEWC =
            new PropertyDescriptor("edgeWeightCutoff",
                class);
    }
    ...

Figure 7.24: Overriding getPropertyDescriptors(): declaring edited properties
and their propertyEditors
...  
// declare logLevel property  
PropertyDescriptor pdLL =  
    new PropertyDescriptor("logLevel",  
        .class);

// (also specify the propertyEditor for LogLevel)  
pdLL.setPropertyEditorClass(socnet.core.LogLevelEditor.class);

// declare autoCalculate property  
PropertyDescriptor pdAC =  
    new PropertyDescriptor("autoCalculate",  
        .class);

// declare autoClearInputs property  
PropertyDescriptor pdACI =  
    new PropertyDescriptor("autoClearInputs",  
        .class);

...  

Figure 7.25: Overriding getPropertyDescriptors(): declaring already-existing properties

    // create an array of the descriptors and return it to  
    // the caller  
    PropertyDescriptor[] pda =  
        {pdSBC, pdBCM, pdLL, pdAC, pdACI, pdEWG};  
        return pda;
    } catch (Exception e) {
        return null;
    }
} // (end of method: getPropertyDescriptors)

} // (end of class: BeanInfo)

Figure 7.26: Overriding getPropertyDescriptors(): returning the array of properties to the caller

299
7.6 Preparation of beans for use in Studio

Making your bean or beans easier to install for users of Geovista Studio involves several steps:

- Pack your bean(s) up into a .jar file
- Predefine a bean palette and a bean folder in Studio
- Include your bean on that palette and bean folder
- Configure your bean(s) for the inputs, output (propertyChange, propertyChange) and callbacks you want to show by default
- Save your palette and bean folder(s) to an XML-based palette file

Click on Palette and Save Palette As as seen in Figure 7.28. Type in the name.

- Distribute the palette file along with your bean

For a tutorial on JAR files, see the online Java Tutorial, Trail on Jar Files (Sommerer, 2003) for help on doing so. Keep in mind that in writing the jar file’s manifest, you should add a line which declares which classes are beans. An example may be seen in Figure 7.27.

Each time you test your bean, you should package it up into a .jar file and put it where your JVM can find it (such as in the extensions directory, lib/ext, as detailed in Section 2.2.2). Then you should import it again into Studio using Palette | New Bean.

When you are ready to distribute your bean, you should distribute it on its own palette, which should have its own palette file.
For the convenience of your users, you should not have them use the manual method of importing beans outlined for the ChemSymphony beans in Section 2.5. Instead, since you are providing them with a palette file for your beans, and a .jar file, they should use the more automatic method. That is, they should simply install the .jar file into their JVM’s extensions directory, as you did (as detailed in Section 2.2.2), and then import your palette file (in much the same way as you installed my socnet.xml palette file in Section 2.4.3).

... 

Name: socnet/jastern/rw/ReWeight.class
Java-Bean: true

Name: socnet/jastern/rw/ReWeightBeanInfo.class
Java-Bean: false

...

Figure 7.27: A snippet of the socnet_jastern.jar manifest

The current version of Geovista Studio does have the current restriction that you may not specify that your bean be on a previously-existing palette. That is, there is one palette file per palette. This leaves your naming convention up to you.

7.7 Using FunctionBean in Java source code

While the focus of this project has been on visual use of the beans in Studio, they may of course also be used in programmatic environments. An example may be seen in Figure 7.29. This example shows the simplicity of using the Socnet beans programmatically. Three beans, TensorToGraphArray, GraphArray-
ToMage, and StringToFile, are created and used to take a Tensor (tns1), convert it to a grapharray, then to a kinemage (in form of a string), and then save the kinemage to a file for later reading with Mage.

More complex examples, including use in multi-threaded environments, are provided with the source code.

7.8 Programming with the Data Types

7.8.1 Introduction: The search for matrix and graph data types

In order to program the beans for this project, several support classes were needed which would represent data types. The data types most immediately apparently needed were a datatype for representing three-dimensional matrices, and a datatype for representing an array of graphs. When it became apparent in
... include socnet.jastern.*;
...

// assuming a Tensor, tns1, created earlier...
// convert our tensor to a graph array
t2ga = new TensorToGraphArray();
t2ga.setInputTensor(tns1);
GraphArray ga1 = t2ga.getOutputGraphArray();

// now convert our grapharray to a kinemage
ga2m = new GraphArrayToMage();
ga2m.setInputGraphArray(ga1);
String km1 = ga2m.getOutputKinemage();

// now save our string to a file
stf = new StringToFile();
stf.setOutputFileName("kinemage1.txt");
stf.setInputString(km1);
...

Figure 7.29: Example of running the beans from Java code in a single-threaded environment

2002 that I would need such data types for the socnet beans to use, I began an extensive search of the World Wide Web with the following criteria in mind. The data types both needed to be:

- Java-based
- Open-source
- Free (at least for educational and research institutions)

Furthermore, the matrix data type needed to:
• be three-dimensional, or at least be extensible into an array of itself (an array of matrices)

• include a tested singular-value decomposition algorithm

while the graph data type had to include support for:

• 3-dimensions

• holding more than one graph

• sub-grouping of the nodes of each graph according to any ad-hoc criteria important to the researcher

I was able to find various pieces of software which met one or more of these criteria, but not all. For instance, for the matrix data type, the Java 3D API (Sun Microsystems Inc, 2003b) looked promising. Its GMatrix class had an SVD method which looked like it would fill the bill but later turned out to have serious errors. IBM’s Alphaworks project had their Graph Foundation Classes for Java (International Business Machines Inc, 2003b). These classes appeared to have many good features, not the least of which was their ease of use. On the other hand, using the classes beyond 90 days required an expensive and restrictive license. Both the Graphlet (Brandenburg, 1999) and Graphviz (AT&T Inc, 2000), (Gansner and North, 1999) programming tools (and languages) were powerful and had non-restrictive licenses, yet were limited to depicting 2-dimensional graphs, and Graphviz furthermore was not in Java; and so on.

Eventually it became apparent that the classes I was looking for simply did not exist. However, it might be possible to build the classes I needed without necessarily starting from scratch. After all, part of the point of this project was to build upon the work of others.
In the end, I created my own classes for this project, based upon work of other open-source projects. The first of these is `socnet.core.Tensor` for a 3-dimensional matrix class, and the second is `socnet.core.GraphArray` for a class of object which could hold an array of graphs and handle grouping of their nodes. I also programmed several other support classes to support these. All of these classes were built upon a number of other classes produced by important working groups. `Tensor` and its sub-class, `SMatrix` are built upon classes from JAMA, a basic linear algebra package for Java (National Institute of Standards and Technology and The MathWorks, 2003). `GraphArray` and its subclasses are based upon important and very well thought out code in the JDSL — the Data Structures Library for Java (JDSL Team, 2003). These projects were willing to release their source code either openly, or under certain free conditions for researchers. Furthermore, licensing for both packages was non-restrictive, easy, and free to researchers, so that both end users and programmers alike could continue to use these classes — indeed, to improve and extend them — in a spirit of collaboration, and without worry. End-users can download the necessary .jar files and include them in Studio. Most importantly, programmers can extend their use with more beans. For instance, a programmer could make a bean which implements a Kamada-Kawai spring-embedder (Kamada and Kawai, 1988), (Kamada and Kawai, 1989) on a `GraphArray`, in an open development environment, without worry of licensing infringement.

However, it must be noted that the Socnet Beans project is not dependent upon any single data type. Indeed, while many see data type standards as a solution to the splintering of the programs available in the field, one of the points of this project is to show that modularized, easily integrated code is a better solution. Anyone can invent a more useful, or more efficient version of a three dimensional matrix, and all that would be required would be to make some new
**FunctionBean**-derived translator beans to translate between **Tensor** and the new data type, and the usefulness of the earlier beans is preserved.

However, I needed *something* to work with when starting, and **Tensor** and **GraphArray** have worked well. Below I give basic outlines to their construction, derivation from other projects, and usage. However, for more specific information, please consult the JavaDocs for these data types on the Socnet beans website (http://orion.oac.uci.edu/~jastern/socnetbeans/), and the source code itself.

Table 7.2 lists the classes in the Java package **socnet.core**. These classes comprise the basic support objects used in this project:

Of these classes, **FunctionBean** and **StackLogger** have already been discussed. **FunctionBeanBeanInfo** and **LogLevelEditor** are support classes for **FunctionBean** and **StackLogger**. These four classes form the basis for each Socnet bean, as each Socnet bean is a **FunctionBean** and a **StackLogger**. The remaining classes are for the two main data types, **Tensor** and **GraphArray**, and will be outlined below.

### 7.8.2 socnet.bean.Tensor

**Tensors** are used by many of the Socnet beans. **Tensor** implements a three-dimensional matrix of real numbers (doubles in Java). It is built upon a **Vector** of **SMatrix**es. Each **SMatrix** is an extension of **Jama.Matrix**.

#### 7.8.2.1 Labelling and StringList

The rows, columns, and matrices of a **Tensor** are labelled. If these labels are not explicitly set, then they will be given defaults (‘row1’, ‘row2’, etc.)
- socnet.core.AttribGroup
- socnet.core.FunctionBeanBeanInfo
- socnet.core.FunctionBean
- socnet.core.GraphArray
- socnet.core.GroupedGraph
- socnet.core.GroupIterator
- socnet.core.GroupTree
- socnet.core LogLevelEditor
- socnet.core.SMatrix
- socnet.core.StackLogger
- socnet.core.StringList
- socnet.core.Tensor

Table 7.2: The socnet.core classes

Tensor provides methods for setting and retrieving labels for its rows, columns and matrices. The row and column labels for one matrix (SMatrix) may not differ from those of the another matrix in the same Tensor. Each instantiation of a Tensor also has a title.

The labels for rows, columns and matrices are all implemented with another class programmed for this project: StringList. StringList provides the necessary methods for creation, changes, and additions. There are even convenient
constructor methods for creation of a **StringList** by shortcuts, such as by provision of the number of labels, a prefix, such as 'row' and a starting number, such as '0', so as to create a **StringList** with the labels (**Strings**), 'row0', 'row1', and so forth.

### 7.8.2.2 Indexing and SMatrix

**Tensor** also provides methods for setting and getting individual cell values via \(x, y, z\) indexing into the row, column, and matrix. However, for speed, **Tensor** also lets you get direct access to the individual **SMatrix**es it contains, so that you may more quickly get or set the values of the cells in that matrix. **Tensor** also provides methods for setting all the cells in the **Tensor** to the same given value (**scale()**), and another similar method for setting all cells to 0 (**zero()**). Finally, **Tensor** may be re-sized dynamically, using a method for this purpose. However, in normal practice this is not used. Instead, one would normally just create a new **Tensor** and copy over the data which is needed by accesses to each **SMatrix**.

**SMatrix** is an extension of the powerful **Jama.Matrix** class put out by the [National Institute of Standards and Technology and The MathWorks](https://www.mathworks.com) (2003). However, **SMatrix** was written for this project to address some of the specific needs of **Tensor** which **Jama.Matrix** was not able to address. **SMatrix** provides methods for getting row, column, and matrix totals easily, as well as getting the grand total of all cells in the matrix. It provides methods (as with **Tensor**) for setting all cells to the same value, or to zero. It provides a **toString()** method for neatly-formatted output of its contents. And it provides methods for obtaining the maximum or minimum of a column or row.
7.8.2.3 Singular Value Decomposition with Tensor

Tensor relies on the proven Jama.SingularValueDecomposition class to do a singular value decomposition. It performs an SVD on each of its SMATRIXes separately. If an SVD on all the matrices at the same time is required, then the CollapseTensor bean may be used to combine the matrices in a Tensor into a single matrix. Then an SVD may be performed and the ExpandTensor bean used to expand out the matrices again. A collapsed Tensor keeps track of what I call its stackpoints, so that when it is expanded back out, it will know how many matrices to re-create and whether to do so vertically or horizontally (default is vertical). Tensor provides a direct method call for doing an SVD on itself, though it calls Jama.SingularValueDecomposition to do the work.

It should be noted that Jama.SingularValueDecomposition was altered for the purposes of this project. As currently downloadable from the JAMA website, Jama.SingularValueDecomposition — as is true of a normal SVD — can only handle a matrix in which \( m \geq n \), or the number of rows is greater than the number of columns. Normally, to decompose a matrix where \( m < n \), one would simply invert the matrix, perform the SVD, and then take the row scores in place of the column scores and visa versa. Since Jama.SingularValueDecomposition did not gracefully handle this situation (even with a Java exception), I programmed this extra facility into Jama.SingularValueDecomposition, and, because of its non-restrictive licensing, I include the entire Jama package (with the improved Jama.SingularValueDecomposition) with the Socnet beans jar file, so that the user gets this extra functionality when doing an SVD on such matrices. Finally, there was one change I had to make to the Jama.Matrix object: One of the fields was declared as private, which prevented direct access to the array holding the data:
Thus I changed the two declarations in \texttt{Jama.Matrix} for the internal array and for the number-of-rows (m) and number-of-columns (n) variables:

\begin{verbatim}
protected double[][] A;
protected int m, n;
\end{verbatim}

from \texttt{private} to \texttt{protected}. Since the only difference between the two types is that \texttt{protected} variables may be accessed by other objects of the same package, and also by sub-classes, but \textit{not} by any other objects, this means that \texttt{Jama.Matrix} could now be extended. This made \texttt{socnet.core.SMatrix} possible.\footnote{The additional code for both \texttt{Jama.SingularValueDecomposition} and \texttt{Jama.Matrix} has been submitted to the Jama project for inclusion in a later release of these classes, should the project’s authors desire it.}

For more detailed information on \texttt{Tensor}, \texttt{SMatrix}, \texttt{StringList}, and the other support classes, please consult the source code for these classes and the source code for the Socnet beans to see examples of their usage.

\subsection*{7.8.3 \texttt{socnet.bean.GraphArray}}

\texttt{GraphArray} implements an array of graphs. The intention is for \texttt{GraphArray} to be a common graph data type which can retain most if not all of the attributes and qualities of other graph formats. In this way \texttt{GraphArray} may be used to translate between the many graph types available. These other types may be represented as Java classes or simply as graph-description languages for saving of a graph in text files. While there are many such languages and several Java classes, this project deals with only two of them: PDB (Protein Data Bank).
used by ChemSymphony (NetGenics Corp, 2002) and many other chemical display programs; and the Mage software’s kinemage format (Richardson and Richardson, 2001). The beans GraphArrayToPDB and GraphArrayToMage translate a GraphArray to each of those formats, respectively. The TensorToGraphArray bean translates a Tensor into a GraphArray based on certain assumptions, or parameters which may be set in the bean.

7.8.3.1 Architecture of GraphArray and the JDSL

Figure 7.30 is a very simplified description of the architecture of a GraphArray. The items with ‘.’ after them are those for which one or more may occur. In this figure may be seen the relationship between the four socnet classes programmed for this project for handling an array of graphs, and the JDSL classes.

```
  socnet.core.GraphArray
    - java.util.Vector
      - socnet.core.GroupedGraph...
      - socnet.core.GroupTree
        - socnet.core.AttribGroup...
      - jdsl.graph.ref.IncidenceListGraph
        - jdsl.graph.api.Vertex...
        - jdsl.graph.api.Edge...
```

Figure 7.30: Simplified structure of a GraphArray

A GraphArray thus consists of a Vector of GroupedGraphs. This is how GraphArray can store more than one graph. This is useful for time-series data sets, or data sets where graphs which are similar in some way are intended to be compared.

Furthermore, suppose that in any one graph, you are looking at dorm data:
group one (in any one graph) is the skaters, group 2 is the political activists, group 3 is the religious folks, group 4 is the partiers, and so on. Suppose you want to in the end be able to "turn on", or view one group at a time, turning off the others temporarily. Or you would like to color the nodes of each group differently, and so on.

This required more than was supplied by the JDSL directly, or by anything else I could find. So I created AttribGroup for this job, as well as GroupIterator, GroupedTree, and GroupedGraph. (See Table 7.30) GroupIterator is not shown here since it is used as a utility to iterate through AttribGroups, in several places. Together, these socnet.core classes supply the functionality of iterating through a group, or class of vertices or a group, or class of edges.

socnet.core.GroupedGraph was created to store both the graph itself (implemented as a jdsl.graph.ref.IncidenceListGraph) as well as a socnet.core.GroupTree. GroupTree contains a set of attribute groups (socnet.core.AttribGroups), organized hierarchically. Together, the GroupTree and the IncidenceListGraph give GroupedGraph the ability to organize its vertices and edges by their membership in sets of attributes. Thus, a graph may contain one set of vertices which belongs to the set of "males" and another subset which belongs to "females". The same is true of edges.

GroupedGraph can hold the same information by virtue of its GroupTree and the IncidenceListGraph. That is, GroupTree is just a tree of groups, and the facility of allowing the programmer to traverse (or iterate) through the nodes or edges of the IncidenceListGraph which are members of each of these groups.

This aggregational ability of GraphArray makes it possible to translate out into the kinemage format which allows for three levels of subsets of nodes. Then
each separate frame in a kinemage translates out to a separate GroupedGraph in the GraphArray.

On the other hand, vertices do not have to have attributes set, other than coordinates (which are required). They can set a color, molecule, or number if they want, but this is not required. It is required of any bean that is going to convert the GraphArray to formats requiring color or molecules to provide these attributes on the fly if these types of information are not contained in the GraphArray vertices themselves.

By default, all the Vertexes are members of one AttribGroup (containing color and other info), and all the Edges belong to another AttribGroup.

Then, after that, the Vertexes may belong to any additional subgroups (AttribGroups), and the same is true of Edges. Thus, any Vertex (or Edge) may belong to more than one subgroup. It is up to the programmer if he/she wants mutually exclusive sub-sets. I wanted to leave this flexibility so that a programmer use the power of this grouping to deal with networked datasets in a very sophisticated way. However, the programmer must be careful to enforce whichever model — overlapping or exclusive attribute sets — they have chosen.

As an aside, it should be noted that the programmer does not have to iterate through groups to get to the Vertexes or Edges. The main IncidenceListGraph itself may be traversed.

7.8.3.2 Use of the JDSL in GraphArray

The GraphArray class is implemented with the help of the JDSL types listed in Table 7.3. Of these, the most notable are the RedBlackTree, which is used in two places. It is used in socnet.core.AttribGroup to store a dictionary of keys and values used for returning references to the stored attributes them-
selves. It is also used in `socnet.core.GroupTree` to retrieve references to the `AttribGroups` held in the `GroupTree`. Also crucial to `GraphArray` are `IncidenceListGraph`, `Edge` and `EdgeIterator`, and `Vertex` and `VertexIterator`. Instantiations of these classes hold the vertices and edges themselves.

```java
jdsl.core.api.Decorable;
jdsl.core.api.InvalidAttributeException;
jdsl.core.api.Locator;
jdsl.core.api.LocatorIterator;
jdsl.core.ref.ComparableComparator;
jdsl.core.ref.HashtableDecorable;
jdsl.core.ref.RedBlackTree;
jdsl.graph.api.Edge;
jdsl.graph.api.Vertex;
jdsl.graph.api.VertexIterator;
jdsl.graph.api.EdgeIterator;
jdsl.graph.ref.IncidenceListGraph;
```

Table 7.3: The JDSL classes used in `GraphArray`

There is one `GroupTree` per `GroupedGraph`, and a `Vector` of `GroupedGraphs` in a `GraphArray`. The `GroupTree` consists of a `jdsl.graph.ref.RedBlackTree` of nodes, each of which is an `AttribGroup`.

While these classes may appear to have a bit of a complicated API at first, they have been well thought out and are highly extensible. For instance, any `Vertex` or `Edge` may have added to it any number of attributes. That is, any object may be stored along with, or "attached to" a `Vertex`. The same is true of an `Edge`. This makes the class architecture extremely flexible in the long-run. The flexibility and considerable planning that went into the `jdsl.graph.ref.IncidenceListGraph`, along with the added ability of the `socnet.core.GroupTree` and `socnet.core.AttribGroup` to track groups of attributes and assign them
to edges and vertices, should make the **GraphArray** an attractive place to start for social network researchers/programmers.

Please consult the Javadocs (API) for the `socnet.core` classes for more specific details. Also the source code for these classes themselves, and the source code for the beans using **GraphArray** may be consulted for examples of the use of these classes in handling social network data.
CHAPTER 8

Conclusion: Summary and Future Directions

This dissertation has represented a proof-of-concept — that it is possible to leverage recent advances in cross-platform languages and modularized coding, combining these with an architecture based on previous data-flow technology, to address several of the problems hindering the learning of, and development and advances in social network research. By creating an extension of a Java bean which handles the housekeeping, event-notification, and data-communications cycle, and by providing data types which beans may share between them, programmers may now have an easier and quicker way to publish their algorithms. Code may be shared more easily, and a common environment for research and teaching agreed upon.

Once again, it should be noted that while data-types are necessary for the beans to communicate, at the same time, the system is data-type free.

The Socnet beans based on the FunctionBean architecture provide a basis for collaborative social network research. Jar-files containing packaged beans, and palettes describing them, may be easily added to Studio. Furthermore, the beans may just as easily be integrated into normal menu-driven programs by their programmers. Or they may be used by web servers wishing to implement social network algorithms for various uses.

The dissertation demonstrated the installation of the software and its use
in an extremely cross-platform graphical development environment, GeoVista Studio, which has many of the features of a traditional modular visualization environment (MVE). Tips were given for programmers wishing to develop their own FunctionBean-extended beans.

Furthermore, in the spirit of collaboration, the project itself built on the important work of other programmers — the JavaCC parser for the ReadDL bean, JDSL and Jama for the data types, GeoVista Studio, and so on.

However, there are a number of ways the system could be much improved:

While not currently part of the Java bean specification, it would be helpful if a data-flow bean or FunctionBean-type specification were agreed upon as part of the Sun Java Community Process (Sun Microsystems Inc, 2003c). It should be apparent to the reader, and certainly to anyone who has used an MVE that flow-based beans such as FunctionBean, while a specialized type of bean, are highly-applicable and useful. Thus, a specification requiring certain agreed-upon mechanisms or extensions to the Java bean for flow-based beans would be helpful. For instance, including the word ”input” and ”output” in the accessor method, and making this a standard, would help graphical bean environments such as Studio to serve up default nodes for these input properties (and default callbacks for the output properties).

Deciding on a uniform standard for bean communication (such as I have done here with the propertyChange.propertyChange event as the notification method, and then a callback to the upstream bean for the output data) would also make development of supportive environments easier.

Finally, agreeing upon a uniform access method for polling the data-flow bean for various information, such as author, URL, email address, version, a list of default visible inputs and outputs/callbacks would also make it easier on users.
Even such as information as the preferred palette and bean folder (or, more
generally, a "major context" and a "minor context", and so on) would facilitate
the inclusion of new beans in bean building environments, putting them in the
right place without needing a corresponding palette and without querying the
user (who may not know).

Extensions such as these would make data-flow (and even other) beans much
easier to use without restricting their use and operation as regular java beans.
These extensions could be made also to FunctionBean as well, but an agreed-
upon specification would be much more powerful.

Another helpful mechanism would be a repository for data-flow beans. Per-
haps the Socnet beans web site (http://orion.oac.uci.edu/~jastern/socnetbeans/)
will eventually have such a repository.

Also helpful to collaboration would be if GeoVista Studio or another future
environment allowed for palette xml files which overlapped, allowing separate
developers to contribute to the same palette or bean folder.

As to the future of the Socnet beans, and of FunctionBeans in general,
they will appear on the Socnet beans web site (http://orion.oac.uci.edu/
~jastern/socnetbeans/), for now. If there is enough interest, it would be nice to
turn them into a SourceForge (Open Source Development Network 2003) project
or even Eclipse (International Business Machines Inc 2003a) project where social
network and other researchers could collaborate in, and benefit from, their contin-
ued development.
Bibliography


Brandenburg, F. J. (1999). *Graphlet: A Toolkit for graph editors and graph algorithms [Computer Software]*. University of Passau Faculty of Mathematics and Computer Science Chair of Theoretical Computer Science, [http://infosun.fmi.uni-passau.de/Graphlet/](http://infosun.fmi.uni-passau.de/Graphlet/)  304


Ojamäe, L. and Hermansson, K. (2000). *Moviemol [Computer software for MS-DOS]*. [http://www.fos.su.se/moviemol.html](http://www.fos.su.se/moviemol.html) [mailto://moviemol@kemi.uu.se](mailto://moviemol@kemi.uu.se) Linköping University and Uppsala University.


Richardson, D. and Richardson, J. (2001). *Kinemage 5.4 data file format*. [ftp://kinemage.biochem.duke.edu/pub/kinfiles/docsDemos/KinFmt.5.4.txt](ftp://kinemage.biochem.duke.edu/pub/kinfiles/docsDemos/KinFmt.5.4.txt)


Sun Microsystems Inc (2003b). *Java 3D API [Computer programming software]*. Sun Microsystems Inc 304


